Proceedings of the 7th Python in Science conference

Gaël Varoquaux, Travis Vaught, Jarrod Millman

SciPy 2008

CONFERENCE

Proceedings of the 7th Python in Science Conference

SciPy Conference – Pasadena, CA, August 19-24, 2008.

Editors: Gaël Varoquaux, Travis Vaught, Jarrod Millman

Contents

Editorial 3
G. Varoquaux, T. Vaught, J. Millman

The State of SciPy 5
J. Millman, T. Vaught

Exploring Network Structure, Dynamics, and Function using NetworkX 11
A. Hagberg, D. Schult, P. Swart

Interval Arithmetic: Python Implementation and Applications 16
S. Taschini

Experiences Using SciPy for Computer Vision Research 22
D. Eads, E. Rosten

The SciPy Documentation Project (Technical Overview) 27
S. Van der Walt

Matplotlib Solves the Riddle of the Sphinx 29
M. Droettboom

The SciPy Documentation Project 33
J. Harrington

Pysynphot: A Python Re-Implementation of a Legacy App in Astronomy 36
V. Laidler, P. Greenfield, I. Busko, R. Jedrzejewski

How the Large Synoptic Survey Telescope (LSST) is using Python 39
R. Lupton

Realtime Astronomical Time-series Classification and Broadcast Pipeline 42
D. Starr, J. Bloom, J. Brewer

Analysis and Visualization of Multi-Scale Astrophysical Simulations Using Python and NumPy 46
M. Turk

Mayavi: Making 3D Data Visualization Reusable 51
P. Ramachandran, G. Varoquaux

Finite Element Modeling of Contact and Impact Problems Using Python 57
R. Krauss

Circuitscape: A Tool for Landscape Ecology 62
V. Shah, B. McRae

Summarizing Complexity in High Dimensional Spaces 66
K. Young

Converting Python Functions to Dynamically Compiled C 70
I. Schnell

unPython: Converting Python Numerical Programs into C **73**
R. Garg, J. Amaral

Editorial

Gael Varoquaux (gael.varoquaux@normalesup.org) – *Neurospin, CEA Saclay , Bât 145, 91191 Gif-sur-Yvette* France
Travis Vaught (tvaught@enthought.com) – *Enthought, Austin* TX USA
Jarrod Millman (millman@berkeley.edu) – *University of California Berkeley, Berkeley* CA USA

The Annual Scipy Conference began in 2002 with a gathering of some extremely bright (and admittedly odd and passionate) folks. They gathered at the Caltech campus in Pasadena, California to discover and share ideas about a compelling approach to scientific computing. These pioneers had been using Python, a dynamic language, to perform and drive their modeling, data exploration and scientific workflows. At that time, Python already had the advantage of a very fast numerics library as well as the flexibility of being interpreted, procedural (if you wanted it to be), and, in a true academic spirit, permissively open source. That gathering 7 years ago yielded many interesting ideas, but more importantly, it crystalized a community and established relationships that persist today. From those inauspicious beginnings, the conference has now grown to a fully international meeting attracting a variety of interests—yet still inspired by the proposition of a humane language interface to fast computation, data analysis and visualization.

This year marked the 7th edition of the conference; however, it is the first edition for which proceedings are to be published. We are thrilled by this new development. These proceedings are the sign of a maturing community not only of developers, but also of scientific users. Python's use as a tool for producing scientific results is, by now, well established. Its status as the subject of research and academic publication is now being recognized and we hope that the proceedings of the SciPy Conference will help communicate how scientists and engineers are using and building upon various Python tools to solve their problems. Communication is indeed paramount to both the software and the scientific community. A scientist needs access to the methods of others; he also needs to get academic credit for his work, which is often measured in publications and citations. A developer needs to be aware of the ongoing software efforts to reduce duplication of effort; he also needs to advertise his work to potential users.

The variety of subjects covered in the talks, and in the following proceedings, is striking because is shows the extent of the use made of these tools in heterogeneous communities. This is the power of open source projects and a key factor to their success. Different actors are able to contribute to the quality of the tools in ways unique to their field and interests, resulting in a whole stack that is much greater than the sum of its parts.

Among the scientific results presented this year, there is a strong presence of the astronomy community—a community that has a long history in software engineering and is an early adopter of Python. Also of note were the inroads that Python has gained in many other contexts, ranging from ecology and neuroimaging, to mechanical engineering and computer vision. As presented at the conference, scientists are attracted to Python by the quality and ease of use of the language itself, and the richness of the existing scientific packages. Another major presence at the conference this year was the more abstract-oriented research fields of mathematics or graph theory. The SAGE project has been very successful at building upon Python to give a consistent package for number theory, or computational mathematics in general. More component-oriented mathematical projects such as NetworkX, for graph theory, or pyinterval, for interval arithmetic, contribute to the stack from a very different perspective than the usual array-manipulating numerical packages. Finally, a sizable fraction of the talks where focused on the current effort to improve the available tools. On the numerical side, the focus is on speed, with some research in compiling part or all of the Python code to fast machine code. We also note a strong effort on the documentation of the tools, where there has been a tremendous amount of work toward filling a historical gap in user docs.

The SciPy Conference has been supported since its creation by the Center for Advanced Computing Research, at CalTech, and Enthought Inc. In addition, this year we were delighted to receive funding from the Python Software Foundation which allowed us to pay for travel, conference fees and accommodation for 10 students. We are very grateful to Leah Jones, of Enthought, and Julie Ponce, of the CACR (Caltech), for their invaluable help in organizing the conference. The conference, like much of the software presented, was truly a global effort and we are hopeful that the impact will continue to grow.

Organizers

Organizing Committee

Conference Chairs

- Jarrod Millman, UC Berkeley (USA)
- Travis Vaught, Enthought (USA)

Tutorial Coordinators

- Fernando Pérez, UC Berkeley (USA)
- Travis Oliphant, Enthought (USA)

Web Masters

- Gaël Varoquaux, Neurospin, CEA - Saclay (France)
- Stéfan van der Walt, University of Stellenbosch (South Africa)

Program Committee

Program Chair

- Gaël Varoquaux, Neurospin, CEA - Saclay (France)

Program Members

- Anne Archibald, McGill University (Canada)
- Matthew Brett, MRC Cognition and Brain Sciences Unit (UK)
- Perry Greenfield, Space Telescope Science Institute (USA)
- Charles Harris, Space Dynamics Laboratory (USA)
- Ryan Krauss, Southern Illinois University Edwardsville (USA)
- Stéfan van der Walt, University of Stellenbosch (South Africa)

Proceedings Reviewers

- Matthew Brett, MRC Cognition and Brain Sciences Unit (UK)
- Ryan Krauss, Southern Illinois University Edwardsville (USA)
- Jarrow Millman, UC Berkeley (USA)
- Gaël Varoquaux, Neurospin, CEA - Saclay (France)
- Stéfan van der Walt, University of Stellenbosch (South Africa)

The State of SciPy

Jarrod Millman (millman@berkeley.edu) – *University of California Berkeley, Berkeley, CA* USA
Travis Vaught (travis@enthought.com) – *Enthought, Austin, TX* USA

The annual SciPy conference provides a unique opportunity to reflect on the state of scientific programming in Python. In this paper, we will look back on where we have been, discuss where we are, and ask where we are going as a community.

Given the numerous people, projects, packages, and mailing lists that make up the growing SciPy community, it can be difficult to keep track of all the disparate developments. In fact, the annual SciPy conference is one of the few events that brings together the community in a concerted manner. So it is, perhaps, appropriate that we begin the conference proceedings with a paper titled "The State of SciPy". Our hope is we can help provide the context for the many interesting and more specific papers to follow. We also aim to promote a much more detailed discussion of the state of the project, community, and software stack, which will continue throughout the year.

The last year has seen a large number of exciting developments in our community. We have had numerous software releases, increased integration between projects, increased test coverage, and improved documentation. There has also been increased focus on improving release management and code review. While many of the papers in the proceedings describe the content of these developments, this paper attempts to focus on the view from 10,000 feet.

This paper is organized in three sections. First, we present a brief and selective historical overview. Second, we highlight some of the important developments from the last year. In particular, we cover the status of both NumPy and SciPy, community building events, and the larger ecosystem for scientific computing in Python. Finally, we raise the question of where the community is heading and what we should focus on during the coming year. The major goal of the last section is to provide some thoughts for a roadmap forward—to improve how the various projects fit together to create a more unified user environment.

In addition to this being the first year that we have published conference proceedings, it is also the first time we have had a formal presentation on the state of SciPy. It is our hope that these will both continue in future conferences.

Past: Where we have been

Before highlighting some of the communities' accomplishments this year, we briefly present a history of scientific computing in Python. Since almost the first release of Python, there has been interest in the scientific community for using Python. Python is an ideal choice for scientific programming; it is a mature, robust, widely-used, and open source language that is freely distributable for both academic and commercial use. It has a simple, expressive, and accessible syntax. It does not impose a single programming paradigm on scientists but allows one to code at many levels of sophistication, including Matlab style procedural programming familiar to many scientists. Python is available in an easily installable form for almost every software platform, including Windows, Macintosh, Linux, Solaris, FreeBSD and many others. It is therefore well suited to a heterogeneous computing environment. Python is also powerful enough to manage the complexity of large applications, supporting functional programming, object-oriented programming, generic programming, and metaprogramming. In contrast, commercial languages like Matlab and IDL, which also support a simple syntax, do not scale well to many complex programming tasks. Lastly, Python offers strong support for parallel computing. Because it is freely available, and installed by default on most Unix machines, Python is an excellent parallel computing client.

Using Python allows us to build on scientific programming technologies that have been under active development and use for over 10 years; while, at the same time, it allows us to use mixed language programming (primarily C, C++, FORTRAN, and Matlab) integrated under a unified Python interface. IPython (ipython.scipy.org) is the de facto standard interactive shell in the scientic computing community. It has many features for object introspection, system shell access, and its own special command system for adding functionality when working interactively. It is a very efficient environment both for Python code development and for exploration of problems using Python objects (in situations like data analysis). Furthermore, the IPython has support for interactive parallel computing. Matplotlib (matplotlib.sourceforge.net) is a tool for 2D plots and graphs, which has become the standard tool for scientific visualization in Python. NumPy (numpy.scipy.org) is a high-quality, fast, stable package for N-dimensional array mathematics. SciPy (scipy.org) is a collection of Python tools providing an assortment of basic scientific programming algorithms (e.g., statistics, optimization, signal processing, etc.). The combination of IPython, matplotlib, NumPy, and SciPy forms the basis of a Matlab-like environment that provides many of the strengths of Matlab (platform independence, simple syntax, high level algorithms, and visualization routines) without its limitations (proprietary, closed source with a weak object model and limited networking capabilities).

Here is a selective timeline:

- 1994 — Python Matrix object (Jim Fulton)

- 1995 — Numeric born (Jim Hugunin, Konrad Hinsen, Paul Dubois, David Ascher, Jim Fulton)
- 2000 — Numeric moves to sourceforge (Project registered as numpy)
- 2001 — SciPy born (Pearu Peterson, Travis Oliphant, Eric Jones)
- 2001 — IPython born (Fernando Perez)
- 2002 — SciPy '02 - Python for Scientific Computing Workshop
- 2003 — matplotlib born (John Hunter)
- 2003 — Numarray (Perry Greenfield, J. Todd Miller, Rick White, Paul Barrett)
- 2006 — NumPy 1.0 Released

Present: Where we are

With the release of NumPy 1.0 in 2006, the community had a new foundation layer for scientific computing built on the mature, stable Numeric codebase with all the advanced functionality and features developed in Numarray. Of course, the existing scientific software had to be ported to NumPy. While Travis Oliphant spent a considerable effort to ensure that this would be as simple as possible, it did take some time for all the various projects to convert.

This is the second conference for the community since the release of NumPy. At this point, most projects have adopted NumPy as their underlying numeric library.

NumPy and SciPy packages

For several months leading up to last year's conference, we were in the unfortunate position that the current releases of NumPy and SciPy were incompatible. At the conference we decided to resolve this by releasing NumPy 1.0.3.1 and SciPy 0.5.2.1. These releases included a few other minor fixes, but didn't include the bulk of the changes from the trunk. Since then we have had three releases of NumPy and one release of SciPy:

- SciPy 0.6.0 (September 2007)
- NumPy 1.0.4 (November 2007)
- NumPy 1.1.0 (May 2008)
- NumPy 1.1.1 (August 2008)

These releases featured a large number of features, speed-ups, bug-fixes, tests, and improved documentation.

- New masked arrays — *MaskedArray* now subclasses *ndarray*. The behavior of the new *MaskedArray* class reproduces the old one.
- Overhaul of IO Code — The NumPy/SciPy IO code is undergoing a major reworking. NumPy will provide basic IO code for handling NumPy arrays, while SciPy will house file readers and writers for third-party data formats (data, audio, video, image, etc.). NumPy also supports a new standard binary file format (.npy/.npz) for arrays/groups_of_arrays. This is the new default method of storing arrays; pickling arrays is discouraged.
- Better packaging — The win32 installer now solves the previously recurring problem of non-working atlas on different sets of CPU. The new installer simply checks which CPU it is on, and installs the appropriate NumPy accordingly (without atlas if the CPU is not supported). We also now provide an official Universal Mac binary.
- Improved test coverage — This year has seen a concerted focus on better test coverage as well as a push for test-driven development. An increasing number of developers are requesting patches or commits to include unit tests.
- Adopted Python Style Guide [PEP8] — For years the official naming convention for classes in NumPy and SciPy was lower_underscore_separated. Since the official Python convention used CapWords for classes as well as several SciPy-related projects (e.g., ETS, matplotlib), it was confusing and led to both standards being used in our codebase. Going forward, newly created classes should adhere to the Python naming convention. Obviously, we will have to keep some of the old class names around so that we don't needlessly break backward compatibility.

Version numbering. During the lead up to the 1.1.0 release, it became apparent that we needed to become more disciplined in our use of release version numbering. Our current release versioning uses three numbers, separated by periods **<major.minor.bugfix>**.[1] Development code for a new release appends an alphanumeric string to the upcoming release numbers, which designate that status (e.g., alpha, beta) of the development code. For example, here is a key to the current minor 1.2.x release series:

- 1.2.0dev5627 — development version 5627
- 1.2.0a1 - first alpha release
- 1.2.0b2 — second beta release
- 1.2.0rc1 — first release candidate
- 1.2.0 — first stable release
- 1.2.1 — first bug-fix release

According to this numbering scheme, the NumPy 2.0.0 release will be the first in this series to allow us to make more significant changes, which would require large-scale API breaks. The idea being that a major release might require people *rewriting* code where a minor release would require a no more than a small amount of *refactoring*. Bug-fix releases will not require any changes to code depending on our software.

Buildbot. Albert Strasheim and Stéfan van der Walt set up a buildbot for numpy shortly before last year's SciPy conference. The buildbot is an automated system for building and testing. This allows the developers and release manager to better track the ongoing evolution of the software.

Community Involvement

The level of community involvement in the project has seen solid growth over the last year. There were numerous coding sprints, training sessions, and conference events. We also held a number of documentation and bug-fix days. And Gaël Varoquaux set up a SciPy blog aggregator early in 2008, which currently has almost fifteen subscribers:

`http://planet.scipy.org/`

Sprints. Sprints have become popular coding activities among many open-source projects. As the name implies, sprints are essentially short, focused coding periods where project members work together in the same physical location. By bringing developers in same location for short-periods of time allows them to socialize, collaborate, and communicate more effectively than working together remotely. While the SciPy community has had sprints for a number of years, this year saw a marked increase. Here is a list of a few of them:

- August 2007 — SciPy 2007 post-conference sprint
- December 2007 — SciPy sprint at UC Berkeley
- February 2008 — SciPy/SAGE sprint at Enthought
- March 2008 — NIPY/IPython sprint in Paris
- April 2008 — SciPy sprint at UC Berkeley
- July 2008 — Mayavi sprint at Enthought

Conferences. In addition to the SciPy 2008 conference, SciPy has had a major presence in several other conferences this year:

- PyCon 2008 — Travis Oliphant and Eric Jones taught a tutorial session titled "Introduction to NumPy" and another titled "Tools for Scientific Computing in Python". While, John Hunter taught a session titled "Python plotting with matplotlib and pylab".
- 2008 SIAM annual meeting — Fernando Perez and Randy LeVeque organized a 3-part minisymposium entitled "Python and Sage: Open Source Scientific Computing", which were extremely well received and chosen for the conference highlights page.
- EuroSciPy 2008 — The first ever EuroSciPy conference was held in Leipzig, Germany on Saturday and Sunday July 26-27, 2008. Travis Oliphant delivered the keynote talk on the history of NumPy/SciPy. There were about 45 attendees.

The Larger Ecosystem

The NumPy and SciPy packages form the basis for a much larger collection of projects and tools for scientific computing in Python. While many of the core developers from this larger ecosystem will be discussing recent project developments during the conference, we want to selectively highlight some of the exciting developments in the larger community.

Major Releases. In addition to releases of NumPy and SciPy, there have been a number of important releases.

- matplotlib 0.98 — Matplotlib is a core component of the stack, and the 0.98 release contains a rewrite of the transforms code. It also features mathtext without the need of LaTeX installed, and a 500-pages-long user-guide.
- ETS 3 — The Enthought Tool Suite (ETS) is a collection of components developed by Enthought and their partners. The cornerstone on which these tools rest is the Traits package, which provides explicit type declarations in Python; its features include initialization, validation, delegation, notification, and visualization of typed attributes.
- Mayavi 2 — This Mayavi release is a very important one, as Mayavi 2 now implements all the features of the original Mayavi 1 application. In addition, it is a reusable library, useful as a 3D visualization component in the SciPy ecosystem.
- IPython 0.9 — This release of IPython marks the integration of the parallel computing code with the core IPython interactive shell.

Distribution. While the quantity and quality of the many scientific Python package has been one the strengths of our community, a mechanism to easily and simply install all these packages has been a weakness. While package distribution is an area that still needs improvement, the situation has greatly improved this year. For years the major Linux distributions have provided official packages of the core scientific Python projects including NumPy, SciPy, matplotlib, and IPython. Also starting with version 10.5 "Leopard", Mac OS X ships with NumPy 1.0.1 pre-installed. Also, as mentioned above, the NumPy and

[1] The NumPy 1.0.3.1 and SciPy 0.5.2.1 releases being an aberration from this numbering scheme. In retrospect, those releases should have been numbered 1.0.4 and 0.5.3 respectively.

SciPy projects have worked to provide better binary installers for Windows and Mac. This year has also seen a number of exciting efforts to provide a one stop answer to scientific Python software distribution:

- Python Package Index — While there are still many issues involving the wide-spread adoption of setuptools, an increasing number of projects are providing binary eggs on the Python Packaging Index. This means an increasing number of scientists and engineers can easily install scientific Python packages using *easy_install*.
- Python(x,y) — www.pythonxy.com
- EPD — Enthought Python Distribution, www.enthought.com/epd
- Sage — Sage is a Python-based system which aims at providing an open source, free alternative to existing proprietary mathematical software and does so by integrating multiple open source projects, as well as providing its own native functionality in many areas. It includes by default many scientific packages including NumPy, SciPy, matplotlib, and IPython.

Cool New Tools. There were also several useful tools:

- Sphinx — Documentation-generation tool.
- Cython — New Pyrex: mixing statically-typed, compiled code with dynamically-typed code, with a Python-like syntax.

Future: Where are we going?

What will the future hold? - Improved release management. - More regular releases. - Clear policy on API and ABI changes. - Better unification of the existing projects.
In the broader view, our ecosystem would benefit from more project cohesion and common branding, as well as an IDE (integrated development environment), an end-user application that would serve as an entry point to the different technologies.

NumPy and SciPy packages

NumPy 1.2 and SciPy 0.7 are on track to be released by the end of this month. This is the first synchronous release since NumPy last August. SciPy 0.7 will require NumPy 1.2 and both releases require Python 2.4 or greater and feature:

- Sphinx-based documentation — This summer saw the first NumPy Documentation Marathon, during which many thousands of lines of documentation were written. In addition, a web framework was developed which allows the community to contribute docstrings in a wiki-like fashion, without needing access to the source repository. The new reference guide, which is based on these contributions, was built using the popular Sphinx tool. While the documentation coverage is now better than ever, there is still a lot of work to be done, and we encourage interested parties to register and contribute further.
- Guide to NumPy — Travis Oliphant released his "Guide to NumPy" for free and checked it into the trunk. Work has already begun to convert it to the ReST format used by Sphinx.
- Nose-based testing framework — The NumPy test framework now relies on the nose testing framework version 0.10 or later.

In addition, SciPy 0.7 includes a whole host of new features and improvements including:

- Major sparse matrices improvements —
- Sandbox removed — The sandbox was originally intended to be a staging ground for packages that were undergoing rapid development during the port of SciPy to NumPy. It was also a place where broken code could live. It was never intended to stay in the trunk this long and was finally removed.
- New packages and modules — Several new packages and modules have been added including constants, radial basis functions, hierarchical clustering, and distance measures.

Python 3.0. Python 2.6 and 3.0 should be released before the end of the year. Python 3.0 is a new major release that breaks backward compatibility with 2.x. The 2.6 release is provided to ease forward compatibility. We will need to keep supporting Python 2.x for the at least the near future. If needed, once released we will provide bug-fix releases to the current releases of NumPy and SciPy to ensure that they run on Python 2.6. We don't currently have a time-frame for Python 3.0 support, but we would like to have a Python 3.0 compatible releases before next year's SciPy conference. The [PEP3000] recommends:

1. You should have excellent unit tests with close to full coverage.
2. Port your project to Python 2.6.
3. Turn on the Py3k warnings mode.
4. Test and edit until no warnings remain.
5. Use the 2to3 tool to convert this source code to 3.0 syntax. **Do not manually edit the output!**
6. Test the converted source code under 3.0.
7. If problems are found, make corrections to the **2.6** version of the source code and go back to step 3.
8. When it's time to release, release separate 2.6 and 3.0 tarballs (or whatever archive form you use for releases).

Release Management

While we were able to greatly improve the quality of our releases this year, the release process was a much less than ideal. Features and API-changes continually creep in immediately before releases. Release schedules repeatedly slipped. And the last SciPy release is out of date compared to the trunk. Obviously these problems are not unique to our project. Software development is tricky and requires balancing many different factors and a large distributed project like ours has the added complexity of coordinating the activity of many different people. During the last year, we had several thoughtful conversations about how to improve the situation. A major opportunity for our project this year will be trying to improve the quality of our release management. In particular, we will need to revisit issues involving release management, version control, and code review.

Time Based Releases. Determining when and how to make a new release is a difficult problem for software development projects. While there are many ways to decide when to release code, it is common to think in terms of feature- and time-based releases:

[Feature-based]

A release cycle under this model is driven by deciding what features will be in the next release. Once all the features are complete, the code is stabilized and finally a release is made. Obviously this makes it relatively easy to predict what features will be in the next release, but extremely difficult to determine when the release will occur.

[Time-based]

A release cycle under this model is driven by deciding when the next release will be. This, of course, makes predicting when the release will be out extremely easy, but makes it difficult to know exactly what features will be included in the release.

Over the last several years, many large, distributed, open-source projects have moved to time-based release management. There has been a fair amount of interest among the SciPy development community to move in this direction as well. Time-based releases are increasingly seen as an antidote to the issues associated with more feature driven development in distributed, volunteer development projects (e.g., lack of planning, continual release delays, out of date software, bug reports against old code, frustration among developers and users). Time-based releases also allows "a more controlled development and release process in projects which have little control of their contributors and therefore contributes to the quality of the output" [Mic07]. It also moves the 'release when it's ready' policy down to the level of specific features rather than holding the entire code base hostage.

A major objective of time-based releases is regular releases with less time between releases. This rapid pace of regular releases, in turn, enables more efficient developer coordination, better short and long term planning, and more timely user feedback, which is more easily incorporated into the development process. Time-based releases promote more incremental development, while they discourage large-scale modifications that exceed the time constraints of the release cycle.

An essential feature of moving to time-based releases is determining the length of the release cycle. With a few notable exceptions (e.g., Linux kernel), most projects have followed the GNOME 6-month release cycle, originally proposed by Havoc Pennington [Pen02]. In order to ensure that the project will succeed at meeting a 6-month time frame requires introducing new policies and infrastructure to support the new release strategy. And the control mechanisms established by those policies and infrastructure have to be enforced.

In brief, here is a partial list of issues we will need to address in order to successfully move to time-based release schedule:

- Branching — In order to be able to release on schedule requires that the mainline of development (the trunk) is extremely stable. This requires that a significant amount of work being conducted on branches.
- Reviewing — Another important way to improve the quality of project and keep the trunk in shape is to require peer code review and consensus among the core developers on which branches are ready to be merged.
- Testing — A full test suite is also essential for being able to regularly release code.
- Reverting — Sticking to release schedule requires occasionally reverting commits.
- Postponing — It also requires postponing branch merges until the branch is ready for release.
- Releasing — Since time-based release management relies on a regular releases, the cost of making a release needs to be minimized. In particular, we need to make it much, much easier to create the packages, post the binaries, create the release notes, and send out the announcements.

An important concern when using time-based releases in open-source projects is this very ability to work relatively "privately" on code and then easily contribute it back to the trunk when it is finally ready.

Given how easy it is for developers to create their own private branches and independently work on them using revision control for as long as they wish, it is important to provide social incentives to encourage regular collaboration and interaction with the other members of the project. Working in the open is a core value of the open-source development model. Next we will look at two mechanisms for improving developer

Proposals. Currently there is no obvious mechanism for getting new features accepted into NumPy or SciPy. Anyone with commit access to the trunk may simply start adding new features. Often, the person developing the new feature, will run the feature by the list. While this has served us to this point, the lack of a formal mechanism for feature proposals is less than ideal.

Python has addressed this general issue by requiring proposals for new features conform to a standard design document called a Python Enhancement Proposal (or "PEP"). During the last year, several feature proposals were written following this model:

- A Simple File Format for NumPy Arrays [NEP1]
- Implementing date/time types in NumPy
- Matrix Indexing
- Runtime Optimization
- Solvers Proposal

Code Review. We also lack a formal mechanism for code review. Developers simply commit code directly to the trunk. Recently there has been some interest in leveraging the Rietveld Code Review Tool developed by Guido van Rossum [Ros08]. Rietveld provides web-based code review for subversion projects. Another option would be to use bzr and the code review functionality integrated with Launchpad.

Code review provides a mechanism for validating design and implementation of patches and/or branches. It also increases consistency in design and coding style.

Distributed Version Control. While subversion provides support for branching and merging, it is not its best feature. The difficult of branch tracking and merging under subversion is pronounced enough that most subversion users shy away from it. Since there is a clear advantage to leverage branch development with time-based releases, we will want to consider using version control mechanisms that provide better branching support.

Distributed Version Control Systems (DVCS), unlike centralized systems such as subversion, have no technical concept of a central repository where everyone in the project pulls and pushes changes. Under DVCS every developer typically works on his own local repository or branch. Since everyone has their own branch, the mechanism of code sharing is merging. Given that with DVCS, branching and merging are essential activities, they are extremely well-supported. This makes working on branches and then merging the code in the trunk only once it is ready extremely simple and easy. DVCS also have the advantage that they can be used off-line. Since anyone can create their own branch from the project trunk, it potentially lowers the barrier to project participation. This has the potential to create a greater culture of meritocracy than traditional central version control systems, which require potential project contributors to acquire "committer" status before gaining the ability to commit code changes to the repository. Finally, DVCS makes it much easier to do private work—allowing you to use revision control for preliminary work that you may not want to publish.

Proposed Release Schedule. Assuming we (1) agree to move to a time-based release, (2) figure out how to continuously keep the trunk in releasable condition, and (3) reduce the manual effort required to make both stable and development releases, we should be able to increase the frequencies of our releases considerably.

	Devel	Stable
Oct	1.3.0b1	1.2.1
Nov	1.3.0rc1	
Nov		1.3.0
Jan	1.4.0a1	1.3.1
Mar	1.4.0b1	1.3.2
Apr	1.4.0rc1	
May	1.4.0rc1	1.3.3

subsystem maintainers, release conference calls, irc meetings, synchronized releases of NumPy and SciPy.

Getting involved

- Documentation
- Bug-fixes
- Testing
- Code contributions
- Active Mailing list participation
- Start a local SciPy group
- Code sprints
- Documentation/Bug-fix Days
- Web design

Acknowledgements

- For reviewing... Stéfan van der Walt
- The whole community...

References

[PEP8] http://www.python.org/dev/peps/pep-0008/

[Mic07] Michlmayr, M. (2007). Quality Improvement in Volunteer Free and Open Source Software Projects: Exploring the Impact of Release Management. PhD dissertation, University of Cambridge.

[Pen02] http://mail.gnome.org/archives/gnome-hackers/2002-June/msg00041.html

[NEP1] http://svn.scipy.org/svn/numpy/trunk/numpy/doc/npy-format.txt

[Ros08] http://code.google.com/appengine/articles/rietveld.html

[PEP3000] http://www.python.org/dev/peps/pep-3000/

Exploring Network Structure, Dynamics, and Function using NetworkX

Aric A. Hagberg (hagberg@lanl.gov) – *Los Alamos National Laboratory, Los Alamos, New Mexico* USA
Daniel A. Schult (dschult@colgate.edu) – *Colgate University, Hamilton, NY* USA
Pieter J. Swart (swart@lanl.gov) – *Los Alamos National Laboratory, Los Alamos, New Mexico* USA

NetworkX is a Python language package for exploration and analysis of networks and network algorithms. The core package provides data structures for representing many types of networks, or graphs, including simple graphs, directed graphs, and graphs with parallel edges and self-loops. The nodes in NetworkX graphs can be any (hashable) Python object and edges can contain arbitrary data; this flexibility makes NetworkX ideal for representing networks found in many different scientific fields.

In addition to the basic data structures many graph algorithms are implemented for calculating network properties and structure measures: shortest paths, betweenness centrality, clustering, and degree distribution and many more. NetworkX can read and write various graph formats for easy exchange with existing data, and provides generators for many classic graphs and popular graph models, such as the Erdos-Renyi, Small World, and Barabasi-Albert models.

The ease-of-use and flexibility of the Python programming language together with connection to the SciPy tools make NetworkX a powerful tool for scientific computations. We discuss some of our recent work studying synchronization of coupled oscillators to demonstrate how NetworkX enables research in the field of computational networks.

Introduction

Recent major advances in the theory of networks combined with the ability to collect large-scale network data has increased interest in exploring and analyzing large networks [New03] [BNFT04]. Applications of network analysis techniques are found in many scientific and technological research areas such as gene expression and protein interaction networks, Web Graph structure, Internet traffic analysis, social and collaborative networks including contact networks for the spread of diseases. The rapid growth in network theory has been fueled by its multidisciplinary impact; it provides an important tool in a systems approach to the understanding of many complex systems, especially in the biological sciences.

In these research areas and others, specialized software tools are available that solve domain-specific problems but there are few open-source general-purpose computational network tools [CN] [OFS08]. NetworkX was developed in response to the need for a well-tested and well-documented, open source network analysis tool that can easily span research application domains. It has effectively served as a platform to design theory and algorithms, to rapidly test new hypotheses and models, and to teach the theory of networks.

The structure of a network, or graph, is encoded in the edges (connections, links, ties, arcs, bonds) between nodes (vertices, sites, actors). NetworkX provides basic network data structures for the representation of simple graphs, directed graphs, and graphs with self-loops and parallel edges. It allows (almost) arbitrary objects as nodes and can associate arbitrary objects to edges. This is a powerful advantage; the network structure can be integrated with custom objects and data structures, complementing any pre-existing code and allowing network analysis in any application setting without significant software development. Once a network is represented as a NetworkX object, the network structure can be analyzed using standard algorithms for finding degree distributions (number of edges incident to each node), clustering coefficients (number of triangles each node is part of), shortest paths, spectral measures, and communities.

We began developing NetworkX in 2002 to analyze data and intervention strategies for the epidemic spread of disease [EGK02] and to study the structure and dynamics of social, biological, and infrastructure networks. The initial development was driven by our need for rapid development in a collaborative, multidisciplinary environment. Our initial goals were to build an open-source tool base that could easily grow in a multidisciplinary environment with users and developers that were not necessarily experts in programming or software engineering. We wanted to interface easily with existing code written in C, C++, and FORTRAN, and to painlessly slurp in large nonstandard data sets (one of our early tests involve studying dynamics on a 1.6 million node graph with roughly 10 million edges that were changing with time). Python satisfied all of our requirements but there was no existing API or graph implementation that was suitable for our project. Inspired by a 1998 essay by Python creator Guido van Rossum on a Python graph representation [vR98] we developed NetworkX as a tool for the field of computational networks. NetworkX had a public premier at the 2004 SciPy annual conference and was released as open source software in April 2005.

In this paper we describe NetworkX and demonstrate how it has enabled our recent work studying synchronization of coupled oscillators. In the following we give a brief introduction to NetworkX with basic examples that demonstrate some of the classes, data structures, and algorithms. After that we describe in detail a research project in which NetworkX plays a central role. We conclude with examples of how others have used NetworkX in research and education.

Using NetworkX

To get started with NetworkX you will need the Python language system and the NetworkX package. Both are included in several standard operating system packages [pac]. NetworkX is easy to install and we suggest you visit the project website to make sure you have the latest software version and documentation [HSS]. In some of the following examples we also show how NetworkX interacts with other optional Python packages such as NumPy, SciPy, and Matplotlib, and we suggest you also consider installing those; NetworkX will automatically use them if they are available.

The basic *Graph* class is used to hold the network information. Nodes can be added as follows:

```
>>> import networkx
>>> G = networkx.Graph()
>>> G.add_node(1) # integer
>>> G.add_node('a') # string
>>> print G.nodes()
['a', 1]
```

Nodes can be any hashable object such as strings, numbers, files, or functions,

```
>>> import math
>>> G.add_node(math.cos) # cosine function
>>> fh = open('tmp.txt','w')
>>> G.add_node(fh) # file handle
>>> print G.nodes()
[<built-in function cos>,
<open file 'tmp.txt', mode 'w' at 0x30dc38>]
```

Edges, or links, between nodes are represented as tuples of nodes. They can be added simply

```
>>> G.add_edge(1,'a')
>>> G.add_edge('b',math.cos)
>>> print G.edges()
[('b', <built-in function cos>), ('a', 1)]
```

When adding an edge, if the nodes do not already exist they are automatically added to the graph.

Edge data d can be associated with the edge by adding an edge as a 3-tuple (u, v, d). The default value for d is the integer 1 but any valid Python object is allowed. Using numbers as edge data allows a natural way to express weighted networks. In the following example we use Dijkstra's algorithm to find the shortest weighted path through a simple network of four edges with weights.

```
>>> G = networkx.Graph()
>>> e = [('a','b',0.3),('b','c',0.9),
       ('a','c',0.5),('c','d',1.2)]
>>> G.add_edges_from(e)
>>> print networkx.dijsktra_path(G,'a','d')
['a', 'c', 'd']
```

NetworkX includes functions for computing network statistics and metrics such as diameter, degree distribution, number of connected components, clustering coefficient, and betweenness centrality. In addition, generators for many classic graphs and random graph models are provided. These graphs are useful for modeling and analysis of network data and also for testing new algorithms or network metrics. The following example shows how to generate and compute some statistics for a network consisting of a path with 6 nodes:

```
>>> G = networkx.path_graph(6)
>>> print G.degree()
[1, 2, 2, 2, 2, 1]
>>> print networkx.density(G)
0.333333333333
>>> print networkx.diameter(G)
5
>>> print networkx.degree_histogram(G)
[0, 2, 4]
>>> print networkx.betweenness_centrality(G)
{0: 0.0, 1: 0.4, 2: 0.6, 3: 0.6, 4: 0.4, 5: 0.0}
```

NetworkX leverages existing Python libraries to extend the available functionality with interfaces to well-tested numerical and statistical libraries written in C, C++ and FORTRAN. NetworkX graphs can easily be converted to NumPy matrices and SciPy sparse matrices to leverage the linear algebra, statistics, and other tools from those packages. For example, to study the eigenvalue spectrum of the graph Laplacian the NetworkX *laplacian()* function returns a NumPy matrix representation. The eigenvalues can be then easily computed using the numpy.linalg sub-package

```
>>> L = networkx.laplacian(G)
>>> print L  # a NumPy matrix
[[ 1. -1.  0.  0.  0.  0.]
 [-1.  2. -1.  0.  0.  0.]
 [ 0. -1.  2. -1.  0.  0.]
 [ 0.  0. -1.  2. -1.  0.]
 [ 0.  0.  0. -1.  2. -1.]
 [ 0.  0.  0.  0. -1.  1.]]
>>> import numpy.linalg
>>> print numpy.linalg.eigvals(L)
[  3.7321e+00   3.0000e+00   2.0000e+00
   1.0000e+00  -4.0235e-17   2.6795e-01]
```

For visualizing networks, NetworkX includes an interface to Python's Matplotlib plotting package along with simple node positioning algorithms based on force-directed, spectral, and geometric methods.

```
>>> G = networkx.circular_ladder_graph(12)
>>> networkx.draw(G)
```

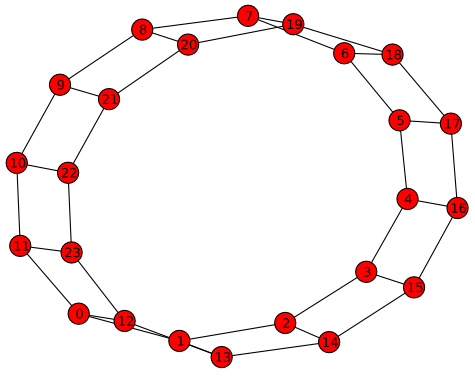


Figure 1: Matplotlib plot of a 24 node circular ladder graph

Connections to other graph drawing packages are available either directly, for example using PyGraphviz with the Graphviz drawing system, or by writing the data to one of the standard file interchange formats.

Inside NetworkX

NetworkX provides classes to represent directed and undirected graphs, with optional weights and self loops, and a special representation for multigraphs which allows multiple edges between pairs of nodes. Basic graph manipulations such as adding or removing nodes or edges are provided as class methods. Some standard graph reporting such as listing nodes or edges or computing node degree are also provided as class methods, but more complex statistics and algorithms such as clustering, shortest paths, and visualization are provided as package functions.

The standard data structures for representating graphs are edge lists, adjacency matrices, and adjacency lists. The choice of data structure affects both the storage and computational time for graph algorithms [Sed02]. For large sparse networks, in which only a small fraction of the possible edges are present, adjacency lists are preferred since the storage requirement is the smallest (proportional to $m + n$ for n nodes and m edges). Many real-world graphs and network models are sparse so NetworkX uses adjacency lists.

Python built-in dictionaries provide a natural data structure to search and update adjacency lists [vR98]; NetworkX uses a "dictionary of dictionaries" ("hash of hashes") as the basic graph data structure. Each node n is a key in the $G.adj$ dictionary with value consisting of a dictionary with neighbors as keys to edge data values with default 1. For example, the representation of an undirected graph with edges $A-B$ and $B-C$ is

```
>>> G = networkx.Graph()
>>> G.add_edge('A','B')
>>> G.add_edge('B','C')
>>> print G.adj
{'A': {'B': 1},
 'B': {'A': 1, 'C': 1},
 'C': {'B': 1}}
```

The outer node dictionary allows the natural expressions *n in G* to test if the graph G contains node n and *for n in G* to loop over all nodes [Epp08]. The "dictionary of dictionary" data structure allows finding and removing edges with two dictionary look-ups instead of a dictionary look-up and a search when using a "dictionary of lists". The same fast look-up could be achieved using sets of neighbors, but neighbor dictionaries allow arbitrary data to be attached to an edge; the phrase $G[u][v]$ returns the edge object associated with the edge between nodes u and v. A common use is to represent a weighted graph by storing a real number value on the edge.

For undirected graphs both representations (e.g $A-B$ and $B-A$) are stored. Storing both representations allows a single dictionary look-up to test if edge $u-v$ or $v-u$ exists. For directed graphs only one of the representations for the edge $u \to v$ needs to be stored but we keep track of both the forward edge and the backward edge in distinct "successor" and "predecessor" dictionary of dictionaries. This extra storage simplifies some algorithms, such as finding shortest paths, when traversing backwards through a graph is useful.

The "dictionary of dictionaries" data structure can also be used to store graphs with parallel edges (multigraphs) where the data for $G[u][v]$ consists of a list of edge objects with one element for each edge connecting nodes u and v. NetworkX provides the $MultiGraph$ and $MultiDiGraph$ classes to implement a graph structure with parallel edges.

There are no custom node objects or edge objects by default in NetworkX. Edges are represented as a two-tuple or three-tuple of nodes (u, v), or (u, v, d) with d as edge data. The edge data d is the value of a dictionary and can thus be any Python object. Nodes are keys in a dictionary and therefore have the same restrictions as Python dictionaries: nodes must be hashable objects. Users can define custom node objects as long as they meet that single requirement. Users can define arbitrary custom edge objects.

NetworkX in action: synchronization

We are using NetworkX in our scientific research for the spectral analysis of network dynamics and to study synchronization in networks of coupled oscillators [HS08]. Synchronization of oscillators is a fundamental problem of dynamical systems with applications to heart and muscle tissue, ecosystem dynamics, secure communication with chaos, neural coordination, memory and epilepsy. The specific question we are investigating is how to best rewire a network in order to enhance or decrease the network's ability to synchronize. We are particularly interested in the setting where the number of edges in a network stays the same while modifying the network by moving edges (defined as removing an edge between one pair of nodes and adding an edge between another). What are the network properties that seriously diminish or enhance synchronization and how hard is it to calculate the required rewirings?

Our model follows the framework presented by [FJC00] where identical oscillators are coupled in a fairly general manner and said to be synchronized if their states are identical at all times. Small perturbations from synchronization are examined to determine if they grow or decay. If the perturbations decay the system is said to be synchronizable. In solving for the growth rate of perturbations, it becomes apparent that the dynamical characteristics of the oscillator and coupling separate from the structural properties of the network over which they are coupled. This surprising and powerful separation implies that coupled oscillators synchronize more effectively on certain networks independent of the type of oscillator or form of coupling.

The effect of the network structure on synchronization is determined via the eigenvalues of the network Laplacian matrix $L = D - A$ where A is the adjacency matrix representation of the network and D is a diagonal matrix of node degrees. For a network with N oscillators, there are N eigenvalues which are all real and non-negative. The lowest $\lambda_0 = 0$ is always zero

and we index the others λ_i in increasing order. For a connected network it is true that $\lambda_i > 0$ for $i > 0$. The growth rate of perturbations is determined by a Master Stability Function (MSF) which takes eigenvalues as inputs and returns the growth rate for that eigenmode. The observed growth rate of the system is the maximum of the MSF evaluations for all eigenvalues. The separation comes about because the MSF is determined by the oscillator and coupling but not by the network structure which only impacts the inputs to the MSF. So long as all eigenvalues lie in an interval where the MSF is negative, the network is synchronizable. Since most oscillator/couplings lead to MSFs where a single interval yields negative growth rates, networks for which the eigenvalues lie in a wide band are resistant to synchronization. An effective measure of the resistance to synchronization is the ratio of the largest to smallest positive eigenvalue of the network, $r = \lambda_{N-1}/\lambda_1$. The goal of enhancing synchronization is then to move edges that optimally decrease r.

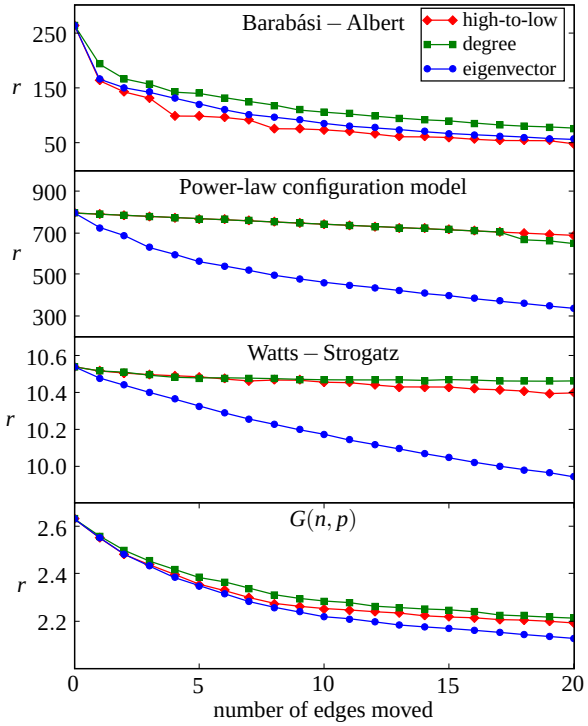


Figure 2: The change in resistance to synchrony r as edges are moved in four example random network models. An algorithm using Laplacian eigenvectors compares favorably to those using node degree. Eigenvectors are found via NetworkX calls to SciPy and NumPy matrix eigenvalue solvers.

Python makes it easy to implement such algorithms quickly and test how well they work. Functions that take NetworkX.Graph() objects as input and return an edge constitute an algorithm for edge addition or removal. Combining these gives algorithms for moving edges. We implemented several algorithms using either the degree of each node or the eigenvectors of the network Laplacian and compared their effectiveness to each other and to random edge choice. We found that while algorithms which use degree information are much better than random edge choice, it is most effective to use information from the eigenvectors of the network rather than degree.

Of course, the specific edge to choose for rewiring depends on the network you start with. NetworkX is helpful for exploring edge choices over many different networks since a variety of networks can be easily created. Real data sets that provide network configurations can be read into Python using simple edge lists as well as many other formats. In addition, a large collection of network model generators are included so that, for example, random networks with a given degree distribution can be easily constructed. These generator algorithms are taken from the literature on random network models. The Numpy package makes it easy to collect statistics over many networks and plot the results via Matplotlib as shown in Fig. 2.

In addition to computation, visualization of the networks is helpful. NetworkX provide hooks into Matplotlib or Graphviz (2D) and VTK or UbiGraph (3D) and thereby allow network visualization with node and edge traits that correlate well with r as shown in Fig. 3.

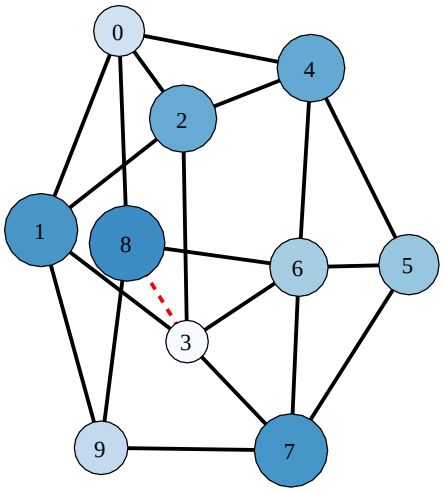


Figure 3: A sample graph showing eigenvector elements associated with each node as their size. The dashed edge shows the largest difference between two nodes. Moving the edge between nodes 3 and 8 is more effective at enhancing synchronization than the edge between the highest degree nodes 3 and 6.

NetworkX in the world

The core of NetworkX is written completely in Python; this makes the code easy to read, write, and document. Using Python lowers the barrier for students and non-experts to learn, use, and develop network algorithms. The low barrier has encouraged contributions from the open-source community and in university educational

settings [MS07]. The SAGE open source mathematics system [Ste08] has incorporated NetworkX and extended it with even more graph-theoretical algorithms and functions.
NetworkX takes advantage of many existing applications in Python and other languages and brings then together to build a powerful analysis platform. For the computational analysis of networks using techniques from algebraic graph theory, NetworkX uses adjacency matrix representations of networks with NumPy dense matrices and SciPy sparse matrices [Oli06]. The NumPy and SciPy packages also provide linear system and eigenvalue solvers, statistical tools, and many other useful functions. For visualizing and drawing, NetworkX contains interfaces to the Graphviz network layout tools [EGK04], Matplotlib (2d) [Hun07] and UbiGraph (3d) [Vel07]. A variety of standard network Models are included for realization and creation of network models and NetworkX can import graph data from many external formats.

Conclusion

Python provides many tools to ease exploration of scientific problems. One of its strengths is the ability to connect existing code and libraries in a natural way that eases integration of many tools. Here we have shown how NetworkX, in conjunction with the Python packages SciPy, NumPy, Matplotlib and connection to other tools written in FORTRAN and C, provides a powerful tool for computational network analysis. We hope to have enticed you to take a look at NetworkX the next time you need a way to keep track of connections between objects.

Acknowledgements

As an open source project this work has significantly benefited from its own international social network of users. We thank the user community for feedback, bug reports, software contributions and encouragement. This work was carried out under the auspices of the National Nuclear Security Administration of the U.S. Department of Energy at Los Alamos National Laboratory under Contract No. DE-AC52-06NA25396 and partially supported by the Laboratory Directed Research and Development Program.

References

[BNFT04] Eli Ben-Naim, Hans Frauenfelder, and Zoltan Torozckai, editors. *Complex Networks*, volume 650 of *Lecture Notes in Physics*. Springer, 2004.

[CN] Gábor Csárdi and Tamás Nepusz. The igraph library. http://cneurocvs.rmki.kfki.hu/igraph/.

[EGK02] Stephen Eubank, Hasan Guclu, V. S. Anil Kumar, Madhav V. Marathe, Aravind Srinivasan, Zoltan Toroczkai, and Nan Wang. Modelling disease outbreaks in realistic urban social networks. *Nature*, page 180, 2002.

[EGK04] J. Ellson, E.R. Gansner, E. Koutsofios, S.C. North, and G. Woodhull. Graphviz and dynagraph – static and dynamic graph drawing tools. In M. Junger and P. Mutzel, editors, *Graph Drawing Software*, pages 127–148. Springer-Verlag, 2004.

[Epp08] David Eppstein. PADS, a library of Python Algorithms and Data Structures, 2008. http://www.ics.uci.edu/eppstein/PADS/.

[FJC00] Kenneth S. Fink, Gregg Johnson, Tom Carroll, Doug Mar, and Lou Pecora. Three coupled oscillators as a universal probe of synchronization stability in coupled oscillator arrays. *Phys. Rev. E*, 61(5):5080 – 90, MAY 2000.

[HS08] Aric Hagberg and Daniel A. Schult. Rewiring networks for synchronization. To appear in *Chaos*, 2008.

[HSS] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. NetworkX. https://networkx.lanl.gov.

[Hun07] John D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science and Engineering*, 9(3):90–95, May/June 2007.

[MS07] Christopher R. Myers and James P. Sethna. Python for education: Computational methods for nonlinear systems. *Computing in Science and Engineering*, 9(3):75–79, 2007.

[New03] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167 – 256, June 2003.

[OFS08] Joshua O'Madadhain, Danyel Fisher, Padhraic Smyth, Scott White, and Yan-Biao Boey. Analysis and visualization of network data using JUNG. http://jung.sourceforge.net/doc/JUNG_journal.pdf, 2008.

[Oli06] Travis E. Oliphant. *Guide to NumPy*. Provo, UT, March 2006.

[pac] Available in Debian Linux and Fink (OSX) package systems.

[Sed02] Robert Sedgewick. *Algorithms in C: Part 5: Graph algorithms*. Addison- Wesley, Reading, MA, USA, third edition, 2002.

[Ste08] William Stein. *Sage: Open Source Mathematical Software (Version 2.10.2)*. The Sage Group, 2008. http://www.sagemath.org

[Vel07] Todd L. Veldhuizen. Dynamic multilevel graph visualization. Eprint arXiv:cs.GR/07121549, Dec 2007.

[vR98] Guido van Rossum. Python Patterns - Implementing Graphs, 1998. http://www.python.org/doc/essays/graphs/

Interval Arithmetic: Python Implementation and Applications

Stefano Taschini (s.taschini@altis.ch) – *Altis Investment Management AG, Poststrasse 18, 6300 Zug* Switzerland

This paper presents the Python implementation of an interval system in the extended real set that is closed under arithmetic operations. This system consists of the lattice generated by union and intersection of closed intervals, with operations defined by image closure of their real set counterparts. The effects of floating-point rounding are accounted for in the implementation. Two applications will be discussed: (1) estimating the precision of numerical computations, and (2) solving non-linear equations (possibly with multiple solutions) using an interval Newton-Raphson algorithm.

Introduction

Consider the following function, an adaptation [R1] of a classic example by Rump [R2]:

$$f(x,y) = (333.75 - x^2)y^6 + x^2(11x^2y^2 - 121y^4 - 2) + 5.5y^8 + x/(2y)$$

Implementing this function in Python is straightforward:

```
>>> def f(x,y):
...     return (
...         (333.75 - x**2)* y**6 + x**2 *
...             (11* x**2 * y**2 - 121 * y**4 - 2)
...         + 5.5 * y**8 + x/(2*y))
```

Evaluating $f(77617, 33096)$ yields

```
>>> f(77617.0, 33096.0)
1.1726039400531787
```

Since f is a rational function with rational coefficients, it is possible in fact to carry out this computation by hand (or with a symbolic mathematical software), thus obtaining

$$f(77617, 33096) = -\frac{54767}{66192} = -0.827396\ldots$$

Clearly, the former result, 1.1726... is completely wrong: sign, order of magnitude, digits. It is exactly to address the problems arising from the cascading effects of numerical rounding that interval arithmetic was brought to the attention of the computing community. Accordingly, this paper presents the Python implementation [R4] of an interval class that can be used to provide bounds to the propagation of rounding error:

```
>>> from interval import interval
>>> print f(interval(77617.0), interval(33096.0))
interval([-3.54177486215e+21, 3.54177486215e+21])
```

This result, with a spread of approximately 7×10^{21}, highlights the total loss of significance in the result. The original motivations for interval arithmetic do not exhaust its possibilities, though. A later section of this papers presents the application of interval arithmetic to a robust non-linear solver finding all the discrete solutions to an equation in a given interval.

Multiple Precision

One might be led into thinking that a better result in computing Rump's corner case could be achieved simply by adopting a multiple precision package. Unfortunately, the working precision required by an arbitrary computation to produce a result with a given accuracy goal is not obvious.

With `gmpy` [R3], for instance, floating-point values can be constructed with an arbitrary precision (specified in bits). The default 64 bits yield:

```
>>> from gmpy import mpf
>>> f(mpf(77617, 64), mpf(33096, 64))
mpf('-4.2949672948273960595e9',64)
```

This result provides absolutely no indication on its quality. Increasing one more bit, though, causes a rather dramatic change:

```
>>> f(mpf(77617, 65), mpf(33096, 65))
mpf('-8.27396059946821368141165095479816292e-1',65)
```

One is still left wandering whether further increasing the precision would produce completely different results.

The same conclusion holds when using the `decimal` package in the standard library.

```
>>> from decimal import Decimal, getcontext
>>> def fd(x,y):
...     return (
...         (Decimal('333.75')-x**2)* y**6 + x**2 *
...             (11* x**2 * y**2 - 121*y**4 - 2)
...         + Decimal('5.5') * y**8 + x/(2*y))
```

The default precision still yields meaningless result:

```
>>> fd(Decimal(77617), Decimal(33096))
Decimal("-999999998.8273960599468213681")
```

In order to get a decently approximated result, the required precision needs to be known in advance:

```
>>> getcontext().prec = 37
>>> fd(Decimal(77617), Decimal(33096))
Decimal("-0.827396059946821368141165095479816292")
```

Just to prevent misunderstandings, the purpose of this section is not to belittle other people's work on multiple-precision floating-point arithmetic, but to warn of a possibly naive use to tackle certain issues of numerical precision loss.

Clearly, very interesting future work can be envisaged in the integration of multiple-precision floating-point numbers into the interval system presented in this paper.

Functions of intervals

Notation-wise, the set of all closed intervals with endpoints in a set X is denoted as

$$\mathbb{I}X = \{[a, b] \mid a, b \in X\}$$

The symbols $\mathbb{R}$ and $\mathbb{R}^*$ denote the set of the real numbers and the extended set of real numbers, $\mathbb{R}^* = \mathbb{R} \cup \{-\infty, +\infty\}$. Let $f([a, b])$ be the image of the closed interval $[a, b]$ under the function f. Real analysis teaches that if the interval is bounded and the function is continuous over the interval, then $f([a, b])$ is also a closed, bounded interval, and, more significantly,

$$f([a, b]) = \left[\min_{x\in[a,b]} f(x), \max_{x\in[a,b]} f(x)\right] \tag{1}$$

Computing the minimum and maximum is trivial if the function is monotonic (see Figure 1), and also for the non-monotonic standard mathematical functions (even-exponent power, cosh, sin, cos...) these are relatively easy to determine.

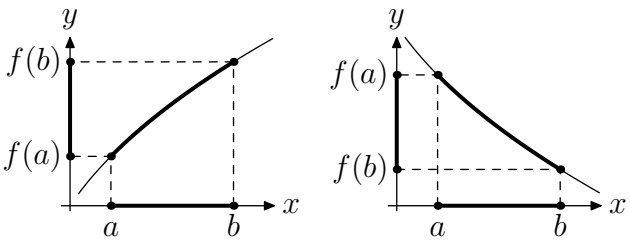


Figure 1. The image $f([a, b])$ for a continuous monotonic function: $[f(a), f(b)]$ for a non-decreasing f (left), and $[f(b), f(a)]$ for a non-increasing f (right).

Equation (1) no longer holds if the interval is unbounded – e.g., $\tanh([0, +\infty]) = [0, 1)$, which is not closed on the right – or the function is not continuous over the whole interval – e.g., the *inverse* function $\mathrm{inv}(x) = 1/x$ yields $\mathrm{inv}([-1, +1]) = (-\infty, -1] \cup [+1, +\infty)$, two disjoint intervals (see Figure 2).

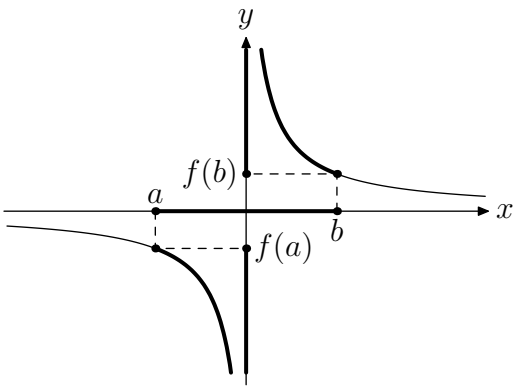


Figure 2. The image $f([a, b])$, with $f(x) = 1/x$, is the union of two disjoint intervals.

Both limitations can be overcome by means of two generalizations: 1) using the image closure instead of the image, and 2) looking at the lattice generated by $\mathbb{IR}^\star$ instead of $\mathbb{IR}^\star$.

The *image closure* is defined for any subset $K \subseteq \mathbb{R}^*$ as

$$\bar{f}(K) = \left\{ \lim_{n\to\infty} f(x_n) \,\middle|\, \lim_{n\to\infty} x_n \in K \right\} \tag{2}$$

Equation (2) is a generalization of equation (1), in the sense that if f is continuous over K and K is a closed, bounded interval, equations (1) and (2) yield the same result, i.e.:

$$f \in C^0([a, b]) \Longrightarrow \bar{f}([a, b]) = f([a, b])$$

The *lattice* generated by the intervals in the extended real set, $L(\mathbb{IR}^*)$, is the smallest family of sets containing $\mathbb{IR}^*$ that is closed under union and intersection – this extension accommodates the fact that, in general, the union of two intervals is not an interval. The sets in the lattice can always be written as the finite union of closed intervals in $\mathbb{R}^*$. In Python,

```
>>> k = interval([0, 1], [2, 3], [10, 15])
```

represents the the union $[0, 1]\cup[2, 3]\cup[10, 15] \in L(\mathbb{IR}^*)$. The intervals $[0, 1]$, $[2, 3]$, and $[10, 15]$ constitute the connected components of `k`. If the lattice element consists of only one component it can be written, e.g., as

```
>>> interval[1, 2]
interval([1.0, 2.0])
```

signifying the interval $[1, 2]$, not to be confused with

```
>>> interval(1, 2)
interval([1.0], [2.0])
```

which denotes $\{1\} \cup \{2\}$. When referring to a lattice element consisting of one degenerate interval, say $\{1\}$, both following short forms yield the same object:

```
>>> interval(1), interval[1]
(interval([1.0]), interval([1.0]))
```

The empty set is represented by an interval with no components:

```
>>> interval()
interval()
```

The state of the art on interval arithmetic [R5] is at present limited to considering either intervals of the form $[a, b]$ with $a, b \in \mathbb{R}^*$ or to pairs $[-\infty, a] \cup [b, \infty]$, as in the Kahan-Novoa-Ritz arithmetic [R6]. The more general idea of taking into consideration the lattice generated by the closed intervals is, as far as the author knows, original.

Note that equation (2) provides a consistent definition for evaluating a function at plus or minus infinity:

$$\bar{f}(\{+\infty\}) = \left\{ \lim_{n\to\infty} f(x_n) \,\middle|\, \lim_{n\to\infty} x_n = +\infty \right\}$$
$$\bar{f}(\{-\infty\}) = \left\{ \lim_{n\to\infty} f(x_n) \,\middle|\, \lim_{n\to\infty} x_n = -\infty \right\}$$

For instance, in the case of the hyperbolic tangent one has that $\overline{\tanh}(\{+\infty\}) = \{1\}$. More generally, it can be proved that if f is discontinuous at most at a finite set of points, then

$$\forall K \in L(\mathbb{IR}^*),\ \bar{f}(K) \in L(\mathbb{IR}^*) \tag{3}$$

The expression in equation (3) can be computed by expressing K as a finite union of intervals, and then by means of the identity

$$\bar{f}\left(\bigcup_h [a_h, b_h]\right) = \bigcup_h \bar{f}([a_h, b_h])$$

For the inverse function, one has that

$$\overline{\mathrm{inv}}\left(\bigcup_h [a_h, b_h]\right) = \bigcup_h \overline{\mathrm{inv}}([a_h, b_h])$$

with

$$\overline{\mathrm{inv}}([a,b]) = \begin{cases} [b^{-1}, a^{-1}] & \text{if } 0 \notin [a,b] \\ [-\infty, \mathrm{inv}_-(a)] \cup [\mathrm{inv}_+(b), +\infty] & \text{if } 0 \in [a,b] \end{cases}$$

where $\mathrm{inv}_-(0) = -\infty$, $\mathrm{inv}_+(0) = +\infty$, and $\mathrm{inv}_-(x) = \mathrm{inv}_+(x) = 1/x$ if $x \neq 0$.
In Python,:

```
>>> interval[0].inverse()
interval([-inf], [inf])
>>> interval[-2,+4].inverse()
interval([-inf, -0.5], [0.25, inf])
```

Interval arithmetic

The definition of image closure can be immediately extended to a function of two variables. This allows sum and multiplication in $L(\mathbb{IR}^*)$ to be defined as

$$H + K = \left\{ \lim_{n\to\infty} (x_n + y_n) \,\middle|\, \lim_{n\to\infty} x_n \in H, \lim_{n\to\infty} y_n \in K \right\}$$
$$H \times K = \left\{ \lim_{n\to\infty} x_n y_n \,\middle|\, \lim_{n\to\infty} x_n \in H, \lim_{n\to\infty} y_n \in K \right\}$$

Since sum and multiplication are continuous in $\mathbb{R}\times\mathbb{R}$ the limits need to be calculated only when at least one of the end-points is infinite. Otherwise the two operations can be computed component-by-component using equation (1). Subtraction and division are defined as

$$H - K = H + \{-1\} \times K$$
$$H \div K = H \times \overline{\mathrm{inv}}(K)$$

These definitions provide a consistent generalization of the real-set arithmetic, in the sense that for any real numbers x and y

$$x \in H, y \in K \Longrightarrow x \diamond y \in H \diamond K$$

whenever $x \diamond y$ is defined, with $\diamond$ representing one of the arithmetic operations. Additionally, this arithmetic is well-defined for infinite end-points and when dividing for intervals containing zero.
In conclusion, the lattice of intervals in the real extended set is closed under the arithmetic operations as defined by image closure of their real counterparts.
In Python, the arithmetic operations are input using the usual `+`, `-`, `*` and `/` operators, with integer-exponent power denoted by the `**` operator. Additionally, intersection and union are denoted using the `&` and `|` operators, respectively.

Dependency

One may not always want to find the image closure of a given function on a given interval. Even for a simple function like $f(x) = x^2 - x$ one might wish to compute $f([0,2])$ by interpreting the expression $x^2 - x$ using interval arithmetic. Interestingly, whereas

$$\forall x \in \mathbb{R},\ x^2 - x = x(x-1) = (x - 1/2)^2 - 1/4$$

the three expressions lead to different results when applied to intervals:

```
>>> (lambda x: x**2 - x)(interval[0,2])
interval([-2.0, 4.0])
>>> (lambda x: x*(x - 1))(interval[0,2])
interval([-2.0, 2.0])
>>> (lambda x: (x - 0.5)**2 - 0.25)(interval[0,2])
interval([-0.25, 2.0])
```

Incidentally, graphic inspection (see Figure 3) immediately reveals that $\bar{f}([0,2]) = [-1/4, 2]$. The three interval functions

$$\begin{aligned} f_1 &: X \in L(\mathbb{IR}^*) \mapsto X^2 - X \\ f_2 &: X \in L(\mathbb{IR}^*) \mapsto X(X-1) \\ f_3 &: X \in L(\mathbb{IR}^*) \mapsto (X - 1/2)^2 - 1/4 \end{aligned} \tag{4}$$

differ because interval arithmetic handles reoccurrences of the same variables as independent instances of the same interval. Only in the case of f_3, where X occurs only once, one has that $f_3(X) = \bar{f}(X)$. For the other two cases, given,

$$\begin{aligned} g_1 &: (x,y) \in \mathbb{R}\times\mathbb{R} \mapsto x^2 - y \\ g_2 &: (x,y) \in \mathbb{R}\times\mathbb{R} \mapsto x(y-1) \end{aligned}$$

one has that $f_1(X) = \bar{g}_1(X, X)$ and $f_2(X) = \bar{g}_2(X, X)$. This phenomenon, called *dependency*, causes f_2 and f_3 to yield in general wider intervals (or the union thereof) than what is returned by the image closure.

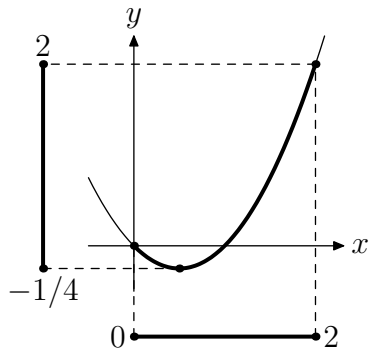


Figure 3. $f([0,2])$ for $f(x) = x^2 - x$.

The idea of a function g on the interval lattice returning "wider" results than needed is captured by saying that g is an *interval extension* of f:

$$g \in \mathrm{ext}(f) \Longleftrightarrow \forall X \in L(\mathbb{IR}^*),\ \bar{f}(X) \subseteq g(X)$$

Referring to the example of equation (4), f_1, f_2, and f_3 are all interval extensions of f. Interval extensions

can be partially ordered by their *sharpness*: given two extensions $g, h \in \mathrm{ext}(f)$, g is *sharper* than h on $X \in L(\mathbb{IR}^*)$ if $g(X) \subset h(X)$.

The extensions f_1, f_2 are not as sharp as f_3 because of dependency. A second source of sharpness loss is rounding, as it will be shown in the following.

Reals and floats

Floating-point numbers, or *floats* in short, form a finite subset $\mathbb{F} \subset \mathbb{R}^*$. It is assumed that floats are defined according to the IEEE 754 standard [R7]. *Rounding* is the process of approximating an arbitrary real number with some float. It is worth noting that rounding is a necessity because for an arbitrary real function f and an arbitrary float $x \in \mathbb{F}$, $f(x)$ is generally not a float. Of the four rounding techniques defined in the standard, relevant for the following are rounding toward $-\infty$, or *down*, defined as

$$\downarrow(x) = \max\{p \in \mathbb{F} \mid p \leq x\}$$

and rounding towards $+\infty$, or *up*, defined as

$$\uparrow(x) = \min\{p \in \mathbb{F} \mid p \geq x\}$$

The interval $I(x) = [\downarrow(x), \uparrow(x)]$ is the *float enclosure* of x, i.e., the smallest interval containing x with end-points in $\mathbb{F}$. The enclosure degenerates to the single-element set $\{x\}$ whenever $x \in \mathbb{F}$. Similarly, for an interval $[a, b]$, its float enclosure is given by $I([a, b]) = [\downarrow(a), \uparrow(b)]$. Note that the enclousure of an interval extension f is also an interval extension, at best as sharp as f.

Also for any of the arithmetic operations, again represented by $\diamond$, it can happen that for any two arbitrary $H, K \in L(\mathbb{IF})$, $H \diamond K \notin L(\mathbb{IF})$. It is therefore necessary to use the float enclosure of the interval arithmetic operations:

$$H \oplus K = I(H + K) \qquad H \ominus K = I(H - K)$$
$$H \otimes K = I(H \times K) \qquad H \oslash K = I(H \div K)$$

In Python, the effect of the float enclosure on the arithmetic operations is easily verifiable:

```
>>> interval[10] / interval[3]
interval([3.333333333333333, 3.3333333333333339])
```

Controlling the rounding mode of the processor's floating-point unit ensures that arithmetic operations are rounded up or down. In the Python implementation presented here, `ctypes` provides the low-level way to access the standard C99 functions as declared in fenv.h [R8], falling back to the Microsoft C runtime equivalents if the former are not present. A lambda expression emulates the lazy evaluation that is required by the primitives in the `interval.fpu` module:

```
>>> from interval import fpu
>>> fpu.down(lambda: 1.0/3.0)
0.33333333333333331
>>> fpu.up(lambda: 1.0/3.0)
0.33333333333333337
```

Unfortunately, common implementations of the C standard mathematical library do not provide the means of controlling how transcendental functions are rounded. For this work it was thus decided to use CR-libm, the Correctly Rounded Mathematical Library [R9], which makes it possible to implement the float enclosure of the image closures for the most common transcendental functions.

The transcendental functions are packaged in the `interval.imath` module:

```
>>> from interval import imath
>>> imath.exp(1)
interval([2.7182818284590451, 2.7182818284590455])
>>> imath.log(interval[-1, 1])
interval([-inf, 0.0])
>>> imath.tanpi(interval[0.25, 0.75])
interval([-inf, -1.0], [1.0, inf])
```

A more compact output for displaying intervals is provided by the `to_s()` method, whereby a string is returned that highlights the common prefix in the decimal expansion of the interval's endpoints. For instance, some of the examples above can be better displayed as:

```
>>> (1 / interval[3]).to_s()
'0.3333333333333333(1,7)'
>>> imath.exp(1).to_s()
'2.718281828459045(1,5)'
```

Solving nonlinear equations

Let f be a smooth function in $[a, b]$, i.e., therein continuous and differentiable. Using the mean-value theorem it can be proved that if $x^* \in [a, b]$ is a zero of f, then

$$\forall \xi \in [a, b], \quad x^* \in \bar{N}(\{\xi\}, [a, b])$$

where N is the Newton iteration function,

$$N(\xi, \eta) = \xi - f(\xi)/f'(\eta) \tag{5}$$

If $f(x) = 0$ has more than one solutions inside $[a, b]$, then, by Rolle's theorem, the derivative must vanish somewhere in $[a, b]$. This in turn nullifies the denominator in equation (5), which causes $\bar{N}(\{\xi\}, [a, b])$ to possibly return two disjoint intervals, in each of which the search can continue. The complete algorithm is implemented in Python as a method of the `interval` class:

```
def newton(self, f, p, maxiter=10000):
    def step(x, i):
        return (x - f(x) / p(i)) & i
    def some(i):
        yield i.midpoint
        for x in i.extrema.components:
            yield x
    def branch(current):
        for n in xrange(maxiter):
            previous = current
            for anchor in some(current):
                current = step(anchor, current)
                if current != previous:
                    break
            else:
                return current
            if not current:
                return current
            if len(current) > 1:
                return self.union(branch(c) for
                       c in current.components)
        return current
    return self.union(branch(c) for
           c in self.components)
```

In this code, `step` implements an interval extension of equation (4), with the additional intersection with the current interval to make sure that iterations are not widening the interval. Function `some` selects ξ: first the midpoint is tried, followed by each of the endpoints. The arguments `f` and `p` represent the function to be nullified and its derivative. The usage of the Newton-Raphson solver is straightforward. For instance, the statement required to find the solutions to the equation

$$(x^2 - 1)(x - 2) = 0 \qquad x \in [-100, +100]$$

simply is

```
>>> interval[-100, 100].newton(
...     lambda x: (x**2 - 1)*(x - 2),
...     lambda x: 3*x**2 - 4*x -1)
interval([-1.0], [1.0], [2.0])
```

Figure 4 shows the iterations needed to solve the same equation in the smaller interval $[-1.5, 3]$. The nonlinear solver can be used with non-algebraic equations as well:

```
>>> interval[-100, 100].newton(
...     lambda x: imath.exp(x) + x,
...     lambda x: imath.exp(x) + 1).to_s()
'-0.567143290409783(95,84)'
```

solves the equation

$$e^x + x = 0 \qquad x \in [-100, +100]$$

and:

```
>>> print interval[-10, 10].newton(
...     lambda x: imath.cospi(x/3) - 0.5,
...     lambda x: -imath.pi * imath.sinpi(x/3) / 3)
interval([-7.0, -7.0], [-5.0, -5.0], [-1.0, -1.0],
         [1.0, 1.0], [5.0, 5.0], [7.0, 7.0])
```

solves the equation

$$\cos\left(\frac{\pi x}{3}\right) = \frac{1}{2} \qquad x \in [-10, +10]$$

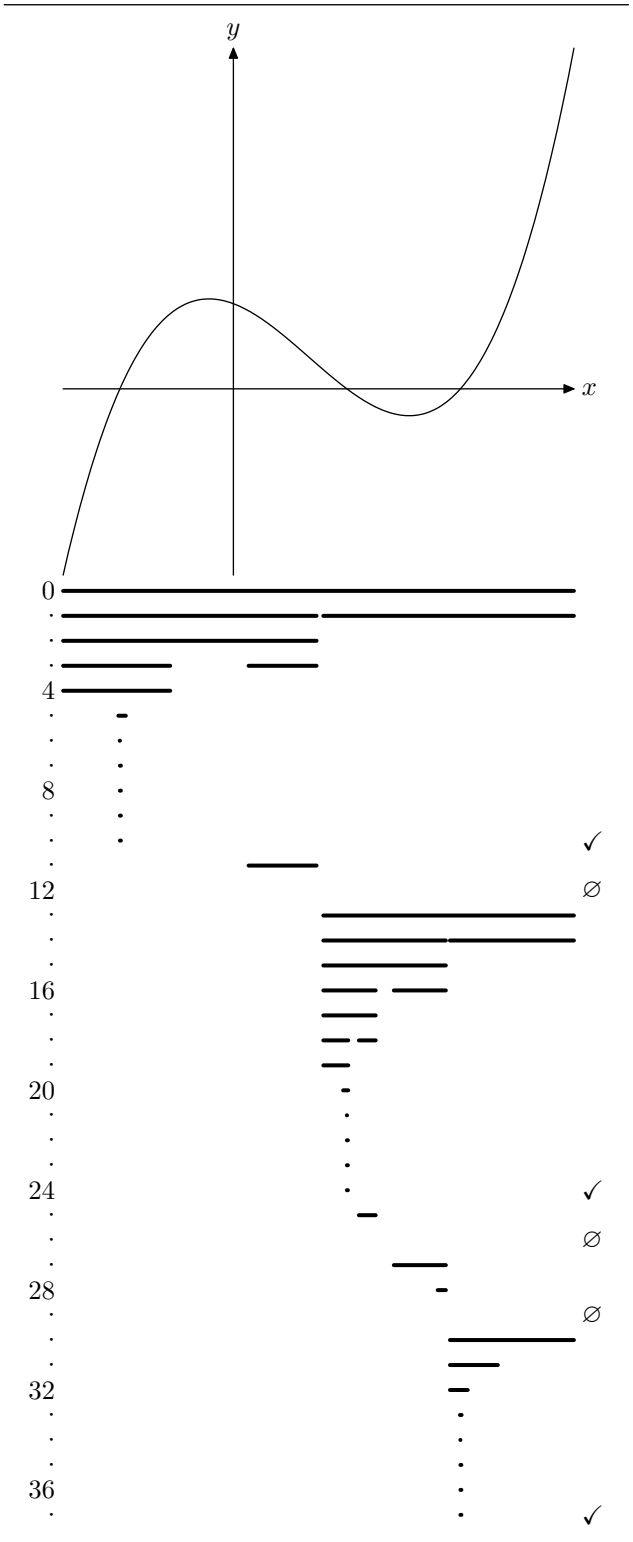


Figure 4. Solving $(x^2 - 1)(x - 2) = 0$ in $[-100, +100]$. An iteration producing an empty interval is marked as ∅, whereas the checkmark denotes an iteration producing a fixed-point.

References

[R1] E. Loh, G. W. Walster, "Rump's example revisited", *Reliable Computing*, vol. 8 (2002), n. 2, pp. 245–248.

[R2] S. M. Rump, "Algorithm for verified inclusions-theory and practice", *Reliability in Computing*, Academic Press, (1988), pp. 109–126.

[R3] A. Martelli (maintainer), "Gmpy, multiprecision arithmetic for Python", http://gmpy.googlecode.com/.

[R4] S. Taschini, "Pyinterval, interval arithmetic in Python", http://pypi.python.org/pypi/pyinterval.

[R5] E. Hansen, G. W. Walster, *Global Optimization Using Interval Analysis*, 2nd edition, John Dekker Inc. (2003).

[R6] R. Baker Kearfott, *Rigorous Global Search: Continuous Problems*, Kluwer (1996).

[R7] D. Goldberg, "What every computer scientist should know about floating-point arithmetic", *ACM Computing Surveys*, vol. 23 (1991), pp. 5–48.

[R8] http://www.opengroup.org/onlinepubs/009695399/basedefs/fenv.h.html.

[R9] J. M. Muller, F. de Dinechin (maintainers), "CR-libm, an efficient and proven correctly-rounded mathematical library", http://lipforge.ens-lyon.fr/www/crlibm/.

Experiences Using SciPy for Computer Vision Research

Damian Eads (eads@lanl.gov) – *Los Alamos National Laboratory, MS D436, Los Alamos, NM* USA
Edward Rosten (edrosten@lanl.gov) – *Los Alamos National Laboratory, MS D436, Los Alamos, NM* USA

SciPy is an effective tool suite for prototyping new algorithms. We share some of our experiences using it for the first time to support our research in object detection. SciPy makes it easy to integrate C code, which is essential when algorithms operating on large data sets cannot be vectorized. The universality of Python, the language in which SciPy was written, gives the researcher access to a broader set of non-numerical libraries to support GUI development, interface with databases, manipulate graph structures, render 3D graphics, unpack binary files, etc. Python's extensive support for operator overloading makes SciPy's syntax as succinct as its competitors, MATLAB, Octave, and R. More profoundly, we found it easy to rework research code written with SciPy into a production application, deployable on numerous platforms.

Introduction

Computer Vision research often involves a great deal of effort spent prototyping new algorithms code. A highly agile, unstructured, iterative approach to code development takes place. Development in low-level languages may be ideal in terms of computational efficiency but is often time consuming and bug prone. MATLAB's succinct "vectorized" syntax and efficient numerical, linear algebra, signal processing, and image processing codes has led to its popularity in the Computer Vision community for prototyping algorithms. Last year, we started a completely new research project in object detection using the SciPy+Python [Jon01] [GvR92] framework without any extensive experience developing with it but having substantial knowhow with MATLAB and C++. The software had to run on Windows and be packaged with an installer. The research problem was very open-ended so a large number of prototype algorithms needed development but eventually the most promising among them had to be integrated into a production application. The project sponsor imposed short deadlines so the decision to use a new framework was high risk as we had to learn the new tool set while keeping the research on pace. Postmortem, we found SciPy to be an excellent choice for both prototyping new code and migrating prototypes into a production system. Acquiring proficiency with SciPy was quick: completing useful, complicated tasks was achievable within a few hours of first installing the software. In this paper, we share some noteworthy reflections on our first experience with SciPy in a full-scale research project.

A Universal Language

One of the strengths of SciPy is that it is a library for Python, a universal and pervasive language. This has two main benefits. First, there is a separation of concerns: the language (Python) is developed independently of the SciPy tool set. The Python community focuses strictly on maintaining the language and its interpreter while the SciPy community focuses on the development of scientific tool sets. The efforts of both groups are not spread thinly across both tasks freeing more time to focus on good design, maintenance, reliability, and support. MATLAB [Mwc82], R [Rcd04], and Octave [Eat02] must instead accomplish several tasks at once: designing a language, implementing and maintaining an interpreter, and developing numerical codes. Second, the universality of Python means there is a much broader spectrum of self-contained communities beyond scientific computation, each of which solely focuses on a single kind of library (e.g. GUI, database, network I/O, cluster computation). Third-party library communities are not as common for highly specialized numerical languages so additional effort must be spent developing tertiary capabilities such as GUI development, database libraries, image I/O, etc. This further worsens the "thin spread" problem: there are fewer time and resources to focus on the two core tasks: developing the language and developing numerical and scientific libraries.

SciPy does not suffer from the "thin spread" problem because of the breadth of libraries available from the many self-contained Python communities. As long as a library is written in Python, it can be integrated into a SciPy application. This was very beneficial to our research in computer vision because we needed capabilities such as image I/O, GUI development, etc. This enabled a more seamless migration into production system.

Operator Overloading: Succinct Syntax

Python's extensive support for operator overloading is a big factor in the success of the SciPy tool set. The array bracket and slice operators give NumPy great flexibility and succinctness in the slicing of arrays (e.g. `B=A[::-1,::-1].T` flips a rectangular array in both directions then transposes the result.)

Slicing an array on the left-hand side of an assignment performs assignments in-place, which is particularly useful in computer vision where data sets are large and unnecessary copying can be costly or fatal. If an array consumes half of the available memory on a machine, an accidental copy will likely result in an `OutOfMemory`

error. This is particularly unacceptable when an algorithm takes several weeks to run and large portions of state cannot be easily check-pointed.

In NumPy, array objects either own their data or are simply a view of another array's data. Both array slicing and transposition generate array views so they do not involve copying. Instead, a new view is created as an array object with its data pointing to the data of the original array but with its striding parameters recalculated.

Extensions

Prior to the project's start, we wrote a large corpora of computer vision code in C++, packaged as the Cambridge Video Dynamics Library (`LIBCVD`) [Ros04]. Since many algorithms being researched depended on these low-level codes, a thorough review of different alternatives for C and C++ extensions in Python was needed. Interestingly, we eventually settled on the native Python C extensions interface after trying several other packages intended to enhance or replace it.

A brief description is given of the `Image` and `ImageRef` classes, the most pervasive data structures within `LIBCVD`. An `Image<T>` object allocates its own raster buffer and manages its deallocation while its subclass `BasicImage<T>` is constructed from a buffer and is not responsible for the buffer's deallocation. The `ImageRef` class represents a coordinate in an image, used for indexing pixels in an `Image` object.

ctypes

`ctypes` [Hel00] seems the easiest and quickest to get started but has a major drawback in that `distutils` does not support compilation of shared libraries on Windows and Mac OS X. We also found it cumbersome to translate templated C++ data structures into NumPy arrays. The data structure would first need to be converted into a C-style array, passed back to Python space, and then converted to a NumPy array. For example, a set of (x, y) coordinates would be represented using `std::vector<ImageRef>` where the coordinates are defined as `struct ImageRef {int x, y;};`. The function for converting a vector of these `ImageRef` structs into a C array is:

```
int *convertToC(vector <ImageRef> &xy_pairs,
                int *num) {
  int *retval = new int[xy_pairs.size()*2];
  *num = xy_pairs.size();
  for (int i = 0; i < xy_pairs.size(); i++) {
    retval[i*2] = xy_pairs[i].x;
    retval[i*2+1] = xy_pairs[i].y;
  }
  return retval;
}
```

Not knowing in advance the size of output buffers is a common problem in scientific computation. In the example, the number of (x, y) pairs is not known *a priori* so the NumPy array cannot be allocated prior to calling the C++ function generating the pairs. One could use the NumPy array allocation function in C++-space but this defeats one of the main advantages of `ctypes`: to interface Python-unaware code.

Once the C-style array is returned back to Python-space, the next natural step is to use the pointer as the data buffer of a new NumPy array object. Unfortunately, this is not easy as it seems because three problems stand in the way. First, there is no function to convert the pointer to a Python buffer object, which is required by the `frombuffer` constructor; second, the `frombuffer` constructor creates arrays that do not own their data; third, even if an array can be created that owns its own data, there is no way to tell NumPy how to deallocate the buffer. In this example, the C++ operator `delete` is required instead of `free()`. In other cases, a C++ destructor would need to be called.

We eventually worked around these issues by creating a Python extension with three functions: one that converts a C-types pointer to a Python buffer object, one that constructs an nd-array that owns its own data from a Python buffer object, and a hook that deallocates memory using C++ `delete`. Even with these functions, each C++ function needs to be wrapped with a C-style equivalent:

```
int* wrap_find_objects(const float *image,
                       int m, int n, int *size) {
  BasicImage <float> cpp(image, ImageRef(m, n));
  vector <ImageRef> cpp_refs;
  find_objects(cpp, cpp_refs);
  *size = cpp_refs.size();
  return convertToC(cpp_refs);
}
```

This function takes in an input image and size, which it converts to a `BasicImage` and calls the `find_objects` routine, which is used to find the (x, y) pairs corresponding to the locations of objects in an image, which it returns as a C-style array. Since `ctypes` does not implement C++ name mangling, there is no function signature embedded in the shared library. Thus, type checking is not performed in Python so a core dump may result when not invoked properly. To avoid these bugs, we needed to create a Python wrapper to do basic type checking of arguments and conversion of input and post-processing of output. `ctypes` is intended to eliminate the need for wrappers, yet two were needed for each C++ function being wrapped. We found `ctypes` inappropriate for our purposes: wrapping large amounts of C++ code safely and efficiently. We did, however, find `ctypes` appropriate for wrapping:

- numerical C codes where the size of output buffers is known ahead of time and can be done in Python-space to avoid ownership and object lifetime issues.
- wrapping non-numerical C codes, particularly those with simple interfaces that use basic C data structures (e.g. encrypting a string, opening a file, or writing a buffer to an image file.)

Weave

With SciPy `weave`, C++ code can be embedded directly in a Python program as a multi-line string in a Python program. MD5 hashes are used to cache compilations of C++ program strings. Whenever the type of a variable is changed, a different program string results, causing a separate compilation. `weave` properly handles iteration over strided arrays. Compilation errors can be cryptic and the translation of a multi-line program string prior to compilation is somewhat opaque. Applications using `weave` need a C++ compiler so it did not fit the requirements of our sponsor. However, we found it useful for quickly prototyping "high risk" for-loop algorithms that could not be vectorized.

Boost Python

Boost Python is a large and powerful library for interfacing C++ code from Python. Learning the tool set is difficult so a large investment of time must be made up front before useful tasks can be accomplished. Boost copies objects created in C++-space, rather than storing pointers to them. This reduces the potential of a dangling reference to an object from Python space, a potentially dangerous situation. Since our computer vision codes often involve large data sets, excessive copying can be a show-stopper.

SWIG

SWIG [Bea95] is a tool for generating C and C++ wrapper code. Our time budget for investigating different alternatives for writing extensions was limited. Several colleagues suggested using the SWIG library to perform the translation and type checking. The documentation of more complicated features is somewhat lacking. The examples are either the "hello world" variety or expert-level without much in between. When deadlines neared, we decided to table consideration of SWIG. However, we encourage those in the SciPy community who have had success with SWIG to document their experiences so others may benefit.

Cython

Cython [Ewi08] is a Python dialect for writing C extensions. Its development has been gaining momentum over the past six months. Python-like code is translated into C code and compiled. It provides support for static type checking as well as facilities for handling object lifetime. Unfortunately, its support for interfacing with templated C++ code is limited. Given the large number of templated C++ functions needing interfacing, it was unsuitable for our purposes.

Native Python C Extensions

As stated earlier, we eventually settled native Python C extensions as our extension framework of choice. A small suite of C++-templated helper functions made the C wrapper functions quite succinct, and performed static type checking to reduce the possibility of introducing bugs.

We found that all the necessary type checking and conversion could be done succinctly in a single C wrapper function and that in most cases, no additional Python wrapper was needed. A few helper functions were written to accommodate the conversion and type checking:

- `BasicImage<T> np2image<T>(img)` converts a rectangular NumPy array with values of type `T` to a `BasicImage` object. If the array does not contain values compatible with `T`, an exception is thrown.
- `PyArrayObject *image2np<T>(img)` converts an `Image` object to a NumPy array of type `T`.
- `PyArrayObject *vec_imageref2np(v)` converts an `std::vector<ImageRef>` of N image references to a N by 2 NumPy array.
- `pair <size_t, T*> np2c(v)` converts a a rectangular NumPy array to a `std::pair` object with the size stored in the first member and the buffer pointer in the second.

Shown below is a boilerplate of a wrapper function. C++ calls go in the `try` block and all errors are caught in the `catch`. All of the helper functions throw an exception if an error occurs during conversion, allocation, or type check. By wrapping the C++ code in a `try/catch`, any C++ exceptions thrown as a `std::string` are immediately caught in the wrapper function. The wrapper function then sets the Python exception string and returns. This solution freed us from having to use Python error handling and NumPy type checking constructs in our core C++ code:

```
PyObject* wrapper(PyObject* self,
                  PyObject* args) {
  try {
    if(!PyArg_ParseTuple(...)) return 0;
    // C++ code goes here.
  }
  catch(string err) {
     PyErr_SetString(PyExc_RuntimeError, err.c_str());
      return 0;
  }
}
```

Shown below is an example of the `np2image` helper function, which converts a `PyArrayObject` to a `BasicImage`. If any errors occur during the type check, a C++ exception is thrown, which gets translated into a Python exception:

```
template<class I>
BasicImage<I> np2image(PyObject *p,
                       const std::string &n="") {

 if (!PyArray_Check(p)
     || PyArray_NDIM(p) != 2
     || !PyArray_ISCONTIGUOUS(p)
     || PyArray_TYPE(p) != NumpyType<I>::num) {
   throw std::string(n + " must be "
     + "a contig array of " + NumpyType<I>::name()
     + "(typecode " + NumpyType<I>::code() + ")!");
 }
 PyArrayObject* image = (PyArrayObject*)p;
 int sm = image->dimensions[1];
 int sn = image->dimensions[0];
 CVD::BasicImage <I> img((I*)image->data,
                         CVD::ImageRef(sm, sn));
 return img;
}
```

Parsing arguments passed to a wrapper function is remarkably simple given a single, highly flexible native extensions function `PyArg_ParseTuple`. It provides basic type checking of Python arguments, such as verifying an object is of type `PyArrayObject`. More thorough type checking of the underlying type of data values in a NumPy array is handled by our C++ helper functions.

Many of the functions in C++ are templated to work across many pixel data types. We needed a solution for passing a NumPy array to an appropriate instance of a templated function without needing a complicated `switch` or `if` statement for each templated function being wrapped. A special wrapper function must be written that generically calls instances of the templated function. We call this a "selector", which is encapsulated in a templated struct:

```
template<class List>
struct convolution_ {
  static PyObject* fun(PyArrayObject *image,
                       double sigma) {
     // Selector code goes here.
  }
}
```

An example of a selector for a convolution function is shown below. It works generically across multiple pixel data types. An instance of the struct is generated for each type in a type list. We iterate through the type list via a form of template-based pattern matching, checking the type of the array with the type of the type in the list's head. If it matches, we call `convolveGaussian`:

```
typedef typename List::type type;
typedef typename List::next next;
if (PyArray_TYPE(image) == NumpyType<type>::num) {
  BasicImage <type> input
    (np2image<type>(image, "image"));
  PyArrayObject *py_result;
  BasicImage <type> result
    (alloc_image<type>(input.size(), &py_result));
  convolveGaussian(input, result, sigma);
  return (PyObject*)py_result;
}
```

Otherwise, we invoke the tail selector:

```
else {
  return _convolution<next>::fun(image, sigma);
}
```

If the type is not supported because none of the types matched, an exception must be thrown:

```
template<>
struct convolution_ <PyCVD::End> {
  static PyObject* fun(PyArrayObject *image,
                       double sigma) {
    throw string("Can't convolve with type: "
                 + PyArray_TYPE(image));
  }
};
```

Finally, the native C wrapper function calls the convolution selector. The selector is templated with a type list of supported types:

```
extern "C" PyObject *convolution(PyObject *self,
                                 PyObject *args) {
  try {
    PyArrayObject *_image;
    double sigma;
    if (!PyArg_ParseTuple(args, "O!d",
             &PyArray_Type, &_image,
             &sigma)) { return 0; }
    return convolution_<CVDTypes>::fun(_image, sigma);
  }
  catch(string err) {
    PyErr_SetString(PyExc_RuntimeError, err.c_str());
    return 0;
  }
}
```

The `TypeList` construct is now defined. All type lists terminate with a special `End` (sentinel) struct:

```
struct End{};
template<class C, class D> struct TypeList {
  typedef C type;  typedef D next;
};
```

Type lists can now be defined to specify supported data types for a selector. The first type list shown is for most numeric data types while the second one is for floating point types:

```
typedef TypeList<char,
                 TypeList<unsigned char,
                 TypeList<short,
                 TypeList<unsigned short,
                 TypeList<int,
                 TypeList<long long,
                 TypeList<unsigned int,
                 TypeList<float,
                 TypeList<double, End>
                 > > > > > > > > CVDTypes;
typedef TypeList<float,
                 TypeList<double, End>
                 > CVDFloatTypes;
```

Lastly, in order to support translation between native C data types and NumPy type codes, we need a macro for defining helper structs to perform the translation:

```
#define DEFNPTYPE(Type, PyType) \
template<> struct NumpyType<Type> {\
  static const int num = PyType; \
  static std::string name(){ return #Type;} \
  static char code(){ return PyType##LTR;} \
}
template<class C> struct NumpyType {};
```

Next, we instantiate a helper struct for each C data type, specifying its corresponding NumPy type code:

```
DEFNPTYPE(unsigned char , NPY_UBYTE );
DEFNPTYPE(char          , NPY_BYTE );
DEFNPTYPE(short         , NPY_SHORT );
DEFNPTYPE(unsigned short, NPY_USHORT);
DEFNPTYPE(int           , NPY_INT   );
DEFNPTYPE(long long     , NPY_LONGLONG);
DEFNPTYPE(unsigned int  , NPY_UINT  );
DEFNPTYPE(float         , NPY_FLOAT );
DEFNPTYPE(double        , NPY_DOUBLE);
```

Comparison with mex

Prior to this project, the authors had some experience working with MATLAB's External Interface (i.e. `mex`). `mex` requires a separate source file for each function. No function exists with the flexibility as Python's `PyArg_ParseTuple`, making it difficult to parse input arguments. Nor does a function exist like `PyBuildValue` to succinctly return data. Opening `mex` files in `gdb` is somewhat cumbersome, making it difficult to pin down segmentation faults. When a framework lacks succinctness and expressibility, developers are tempted to copy code, which often introduces bugs.

Object-oriented Programming

Many algorithms require the use of data structures than other just rectangular arrays, e.g., graphs, sets, maps, and trees. MATLAB's
usually encodes such data structures with a matrix (e.g. the `treeplot` and `etree` functions). MATLAB supports object-oriented programming so in theory one can implement a tree or graph class. The authors have attempted to make use of this facility but found it limited: objects are immutable so changes to them involve a copy. Since computer vision projects typically involve large data sets, if not careful, subtle copying may swamp the system. Moreover, it is difficult to organize a suite of classes because each class must reside in its own directory (named `@classname`) and each method in its own file. Changing the name of a method requires renaming a file and the method name in the file followed by a traversal of all files in the directory to ensure all remaining references are appropriately renamed. Combining three methods into one involves moving the code in the two files into the remaining file and then deleting the originating files. This makes it cumbersome to do agile object-oriented development. To get around these shortcomings, most programmers introduce global variables, but this inevitably leads to bugs and makes code hard to maintain. After many years of trial and error, we found Python+SciPy to be more capable for developing both prototype code and production scientific software.
Python has good facilities for organizing a software library with its *modules* and *packages*. A module is a collection of related classes, functions, and data. All of its members conveniently reside in the same source file. Objects in Python are mutable and all methods of a class are defined in the same source file. Since Python was designed for object-orientation, many subcommunities have created OO libraries to support almost any software engineering task: databases, GUI development, network I/O, or file unpacking. This makes it easy to develop production code.
Data structures such as maps, sets, and lists are built into Python. Python also supports a limited version of a continuation known as a *generator function*, permitting lazy evaluation. Rich data structures such as graphs can easily be integrated into our algorithms by defining a new class. Workarounds such as global variables were not needed. Development with Python's object-oriented interface was remarkably seamless.
In MATLAB, variables are passed by value with copy-on-write semantics. Python's support for pass-by-reference gives one more flexibility by allowing one to pass large arrays to functions and modify them. While these semantics are not as easy to understand as pass-by-value, they are essential for developing production applications as well as for computing on large data sets.

Conclusion

We started a new research project using SciPy without having any previous experience with it. SciPy's succinct, vectorized syntax and its extensive support for slicing makes it a good prototyping framework. The universality of Python gives one access to a wide variety of libraries, e.g. GUI toolkits, database tools, etc., to support production development. Its modules and object-orientation allows for a clean organization of software components. Pass-by-reference semantics permit efficient and safe handling of large data sets. With the flexibility of Python's C extension interface, one can interface with a large corpora of existing C++ code. Our design permits core C++ algorithms to be Python-unaware but with support for error reporting back to the Python environment. Using C++ generics combined with a small suite of macros and helper functions, instances of templated algorithms can be called in a manner that is generic to pixel data type. Overall, we found the Python+SciPy to be an excellent choice to support our research.

References

[Bea95] D. Beazley. Simplified Wrapper and Interface Generator. http://www.swig.org/. 1995-.

[Eat02] J. Eaton. *GNU Octave Manual*. Network Theory Limited Press. 2002.

[Ewi08] M. Ewing. *Cython*. 2008-.

[Hel00] T. Heller. *ctypes*. 2000-.

[Hun02] J. Hunter. *matplotlib: plotting for Python*. http://matplotlib.sf.net/. 2002-.

[Jon01] E. Jones, T. Oliphant, P. Peterson, et al. "SciPy: Open Source Scientific tools for Python". 2001--.

[Mwc82] The Mathworks Corporation. *MATLAB*. 1984-.

[GvR92] G. van Rossum. *Python*. 1991-.

[Rcd04] The R Core Development Team. *R Reference Manual*. 2004-.

[Ros04] E. Rosten, et al. *LIBCVD*. http://savannah.gnu.org/projects/libcvd. 2004-

The SciPy Documentation Project (Technical Overview)

Stefan Johann Van der Walt (stefan@sun.ac.za) – *University of Stellenbosch,* South Africa

This summer saw the first NumPy Documentation Marathon, during which many thousands of lines of documentation were written. In addition, a web framework was developed which allows the community to contribute docstrings in a wiki-like fashion, without needing access to the source repository. The new reference guide, which is based on these contributions, was built using the popular Sphinx tool. While the documentation coverage is now better than ever, there is still a lot of work to be done, and we encourage interested parties to register and contribute further. This paper gives a more detailed overview of the events leading to this project, and of the technical goals achieved.

Motivation and early history

The effort was funded by the University of Central Florida, under management of Dr Joe Harrington. Dr Harrington attempted to use the SciPy tool suite as a basis for his lectures in astronomy last year, but found that students struggled to become proficient with NumPy and SciPy, much more so than with a previous package used. A lack of documentation was identified as the key deficiency, and, in response, this project was started.

Joe hired me in order to write documentation, but we soon realised that the mammoth task of documenting packages as vast as NumPy and SciPy was beyond the means of a single person. We needed to find a way to involve the larger community - to distribute the load of writing and to obtain input from people with a variety of backgrounds.

In March, at the neuro-imaging sprint hosted by NeuroSpin in Paris, the first discussions centred around such a framework took place between Fernando Perez, Gaël Varoquaux and myself. Later that month, at the IPython sprint, Emmanuelle Gouillart volunteered to do the first implementation, and we sketched a rough architecture. The plan was to coerce MoinMoin into serving up docstrings, and propagate changes back into a bzr repository of the NumPy source.

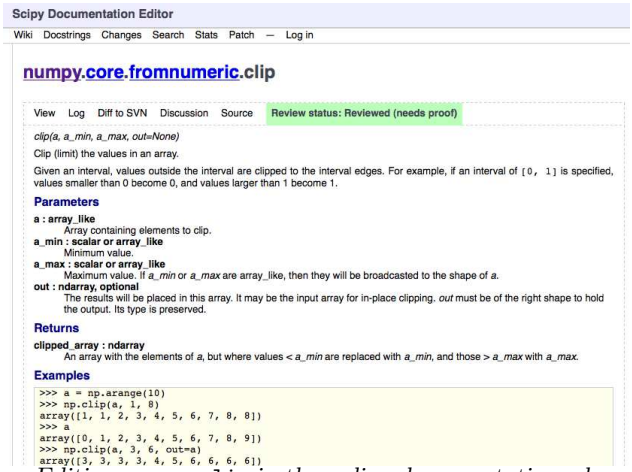
Meanwhile, Pauli Virtanen was hard at work on the same concept, and when he announced his solution to the mailing list, the obvious way forward was to join forces. He and Emmanuelle proceeded to implement the first usable version of the documentation wiki.

Documentation Web-app

The original MoinMoin-based documentation editor that Pauli and Emmanuelle produced provided everything we needed: a way of editing NumPy docstrings in ReStructuredText, and of propagating those changes back to source, albeit with a lot of effort.

Soon, however, it became clear that the MoinMoin-based approach was limited. Adding a feature to the editor often meant crudely hacking it onto a framework written with a different use case in mind. This limited our progress in (and enthusiasm for!) implementing new ideas.

Pauli started working on a web framework in Django, and completed it in record time. The new framework supported all the features of the old one, but, being a web app, allowed us to add things easily such as workflow, merge control, access control, comments and statistics.



Editing `numpy.clip` *in the online documentation editor.*

The source of the web-app is available here:

https://code.launchpad.net/~pauli-virtanen/scipy/pydocweb

Documentation Standard

Even before the online editing infrastructure was completely in place, discussion started on finalising a documentation standard. Since we were going to invest a significant amount of time formatting docstrings, we wanted to do it right from the start. At the time, Jarrod Millman had already written up the format as it stood, and we completed it, aided by feedback from the community.

The documentation standard is tool-agnostic ReStructuredText, and was designed with text terminals in mind. It attempts to balance markup capabilities with ease of readability.

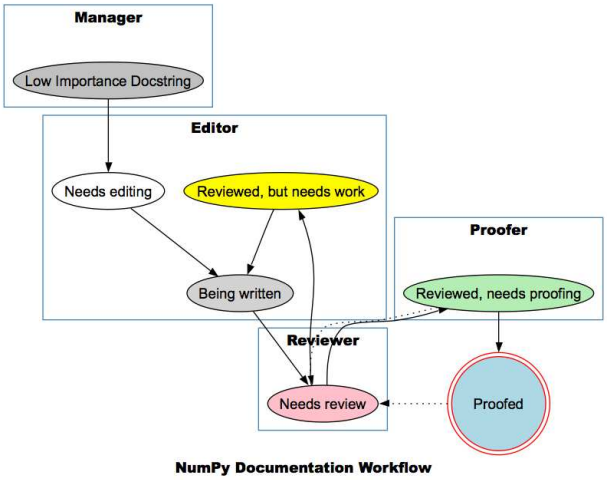
We realised that the new format would not suit all documentation tools, such as Sphinx and Epydoc, and wrote a parser which generates a higher level view of the docstring. It then becomes trivial to output it in any suitable format.

The documentation parser is available at:

https://code.launchpad.net/~stefanv/scipy/numpy-refguide

Workflow

With the new web-app, implementing a workflow became possible:



NumPy Documentation Workflow

The workflow is not rigorously enforced: we focus instead on moving docstrings into the source code as rapidly as possible. The reviewer status was designed to expose the code to more editors and thereby improve the quality of the code, rather than as a mechanism of restricting contribution.

Pauli furthermore implemented three-way merging, which allows us to update docstrings in the web application with those from the Subversion repository. The administrator is given the opportunity to fix any conflicts that may arise. After such an update, edits are made against the latest changes.

Whenever required, a patch can be generated against the latest SVN version of the code. This patch alters our software to contain the latest changes from the wiki, and takes into account special mechanisms for adding docs, such as NumPy's `add_newdocs.py`.

Progress Made

Considering that Dr Harrington funded the documentation drive, the fifty-odd functions used in his classes were given priority. After documenting those, an additional 280 were identified as important targets, of which more than half are now documented.

Writing documentation, particularly examples, leads to a lot of testing, which in turn exposes bugs. The documentation project therefore unexpectedly produced some code fixes to NumPy and Sphinx!

A reference guide of more than 300 pages was generated using the Sphinx tool (which was developed to document Python itself). A Sphinx plugin was written, using the parser mentioned above, to translate the NumPy docstrings to a format ideally suited to Sphinx.

The code used to generate the reference guide can be found either at:

https://code.launchpad.net/~stefanv/scipy/numpy-refguide

or at:

https://code.launchpad.net/~pauli-virtanen/scipy/numpy-refguide

The latest reference guide can be downloaded from:

http://mentat.za.net/numpy/refguide

Conclusion

I would like to thank everyone who contributed to the documentation effort thus far, be it by giving feedback on the mailing list, by coding or by writing documentation. Those of you who wrote more than a thousand words of documentation were rewarded with T-shirt, which is merely a small token of appreciation. In reality, the contribution you have made is an imporant one: significantly lowering a barrier to the adoption of NumPy.

While we have made good progress, there is still a lot left to do! We ask that you join us in writing, reviewing and editing docstrings, or otherwise assist in the ongoing development of the documentation application and tools.

Matplotlib Solves the Riddle of the Sphinx

Michael Droettboom (mdboom@gmail.com) – *Space Telescope Science Institute,* USA

This paper shares our experience converting matplotlib's documentation to use Sphinx and will hopefully encourage other projects to do so. Matplotlib's documentation serves as a good test case, because it includes both narrative text and API docstrings, and makes use of automatically plotted figures and mathematical expressions.

Introduction

Sphinx [Bra08] is the official documentation tool for future versions of Python and uses reStructuredText [Goo06] as its markup language. A number of projects in the scientific Python community (including IPython and NumPy) have also converged on Sphinx as a documentation tool. This standardization, along with the ease-of-use of reStructuredText, should encourage more people to contribute to documentation efforts.

History

Before moving to Sphinx, matplotlib's [Hun08] documentation toolchain was a homegrown system consisting of:

- HTML pages written with the YAPTU templating utility [Mar01], and a large set of custom functions for automatically generating lists of methods, FAQ entries, generating screenshots etc.
- Various documents written directly in LaTeX, for which only PDF was generated.
- pydoc [Yee01] API documentation, only in HTML.
- A set of scripts to build everything.

Moving all of these separate formats and silos of information into a single Sphinx-based build provides a number of advantages over the old approach:

- We can generate printable (PDF) and on-line (HTML) documentation from the same source.
- All documentation is in a single format, reStructuredText, and in plain-text files or docstrings. Therefore, there is less need to copy-paste-and-reformat information in multiple places and risk diverging.
- There are no errors related to manually editing HTML or LaTeX syntax, and therefore the barrier to new contributers is lower.
- The output is more attractive, since the Sphinx developers have HTML/CSS skills that we lack. Also, the docstrings now contain rich formatting, which improves readability over pydoc's raw monospaced text. (See Figures at the end of this paper).
- The resulting content is searchable, indexed and cross-referenced.

Perhaps most importantly, by moving to a standard toolchain, we are able to share our improvements and experiences, and benefit from the contributions of others.

Built-in features

Search, index and cross-referencing

Sphinx includes a search engine that runs completely on the client-side. It does not require any features of a web server beyond serving static web pages. This also means that the search engine works with a locally-installed documentation tree.
Sphinx also generates an index page. While docstrings are automatically added to the index, manually indexing important keywords is inherently labor-intensive so matplotlib hasn't made use of it yet. However, this is a problem we'd like to solve in the long term, since many of the questions on the mailing list arise from not being able to find information that is already documented.

autodoc

Unlike tools like pydoc and epydoc [Lop08], Sphinx isn't primarily a tool for fully-automatic API and code documentation. Instead, its focus is on narrative documentation, meant to be read in a particular order. This difference in bias is not accidental. Georg Brandl, the author of Sphinx, wrote[1]:

> One of Sphinx' goals is to coax people into writing good docs, and that unfortunately involves writing in many instances :) This is not to say that API docs don't have their value; but when I look at a new library's documentation and only see autogenerated API docs, I'm not feeling encouraged.

However, Sphinx does provide special directives to extract and insert docstrings into documentation, collectively called the `autodoc` extension. For example, one can do the following:

```
.. automodule:: matplotlib.pyplot
   :members:
   :show-inheritance:
```

This creates an entry for each class, function, etc. in the `matplotlib.pyplot` module.
There are a number of useful features in epydoc that aren't currently supported by Sphinx including:

[1] In a message on the `sphinx-dev` mailing list on August 4, 2008: http://groups.google.com/group/sphinx-dev/msg/9d173107f7050e63

- Linking directly to the source code.
- Hierarchical tables of modules, classes, methods etc. (Though documented objects are inserted into an alphabetized master index.) This shortcoming is partially addressed by the inheritance diagram extension.
- A summary table with only the first line of each docstring, that links to the complete versions.

In the matplotlib documentation, this last shortcoming is painfully felt by the `pyplot` module, where over one hundred methods are documented at length. There is currently no way to easily browse what methods are available.
Note that Sphinx development progresses rather quickly, and some or all of these shortcomings may be resolved very soon.

Extended features

As Sphinx is written in Python, it is quite easy to write extensions. Extensions can:

- add new builders that, for example, support new output formats or perform actions on the parsed document trees.
- add code triggered by certain events during the build process.
- add new reStructuredText roles and directives, extending the markup. (This is primarily a feature of docutils, but Sphinx makes it easy to include these extensions in your configuration).

Most of the extensions built for matplotlib are of this latter type.
The matplotlib developers have created a number of Sphinx extensions that may be generally useful to the Scientific Python community. Where applicable, these features have been submitted upstream for inclusion in future versions of Sphinx.

Automatically generated plots

Any matplotlib plot can be automatically rendered and included in the documentation. The HTML version of the documentation includes a PNG bitmap and links to a number of other formats, including the source code of the plot. The PDF version of the documentation includes a fully-scalable version of the plot that prints in high quality.
This functionality is very useful for the matplotlib docs, as we can now easily include figures that demonstrate various methods. For example, the following reStructuredText directive inserts a plot generated from an external Python script directly into the document:

```
.. plot:: ../mpl_examples/xcorr_demo.py
```

See Figures for a screenshot of the result.

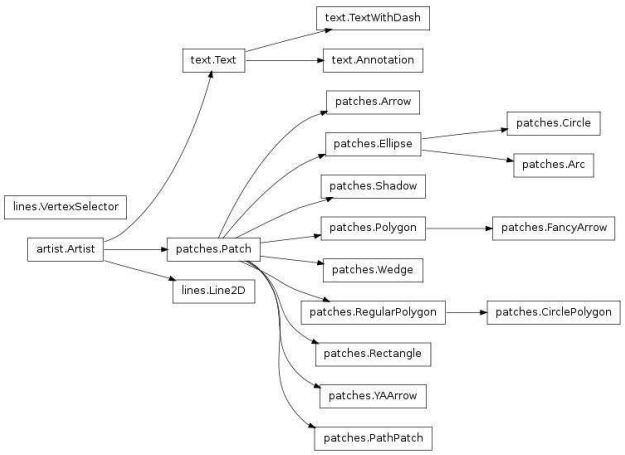
Inheritance diagrams

Given a list of classes or modules, inheritance diagrams can be drawn using the graph layout tool graphviz [Gan06]. The nodes in the graph are hyperlinked to the rest of the documentation, so clicking on a class name brings the user to the documentation for that class.

The reStructuredText directive to produce an inheritance diagram looks like:

```
.. inheritance-diagram:: matplotlib.patches
                         matplotlib.lines
                         matplotlib.text
   :parts: 2
```

which produces:



Mathematical expressions

Matplotlib has built-in rendering for mathematical expressions that does not rely on external tools such as LaTeX, and this feature is used to embed math directly in the Sphinx HTML output.

This rendering engine was recently rewritten by porting a large subset of the TeXmath layout algorithm [Knu86] to Python[2]. As a result, it supports a number of new features:

- radicals, eg., $\sqrt[3]{x}$
- nested expressions, eg., $\sqrt{\frac{x+\frac{1}{3}}{x+1}}$
- wide accents, eg., $\widehat{xyz}$
- large delimiters, eg., $\left(\frac{\delta x}{\delta y}\right) \lfloor z \rfloor$
- support for the STIX math fonts [STI08], giving access to many more symbols than even TeX itself, and a more modern-looking sans-serif math mode.

The following figure shows a complex fictional mathematical expression rendered using the three supported font sets, Computer Modern, STIX and STIX sans serif.

$$W^{3\beta}_{\delta_1\rho_1\sigma_2} = U^{3\beta}_{\delta_1\rho_1} + \sqrt{\frac{1}{8\pi 2}} \int_{\alpha_2}^{\alpha_2} d\alpha_2' \left[\frac{U^{2\beta}_{\delta_1\rho_1} - \alpha_2' U^{1\beta}_{\rho_1\sigma_2}}{\mathbb{A}^{0\beta}_{\rho_1\sigma_2}} \right]$$

$$W^{3\beta}_{\delta_1\rho_1\sigma_2} = U^{3\beta}_{\delta_1\rho_1} + \sqrt{\frac{1}{8\pi 2}} \int_{\alpha_2}^{\alpha_2} d\alpha_2' \left[\frac{U^{2\beta}_{\delta_1\rho_1} - \alpha_2' U^{1\beta}_{\rho_1\sigma_2}}{\mathbb{A}^{0\beta}_{\rho_1\sigma_2}} \right]$$

$$W^{3\beta}_{\delta_1\rho_1\sigma_2} = U^{3\beta}_{\delta_1\rho_1} + \sqrt{\frac{1}{8\pi 2}} \int_{\alpha_2}^{\alpha_2} d\alpha_2' \left[\frac{U^{2\beta}_{\delta_1\rho_1} - \alpha_2' U^{1\beta}_{\rho_1\sigma_2}}{\mathbb{A}^{0\beta}_{\rho_1\sigma_2}} \right]$$

The use of this extension in the matplotlib documentation is primarily a way to test for regressions in our own math rendering engine. However, it is also useful for generating math expressions on platforms that lack a LaTeX installation, particularly on Microsoft Windows and Apple OS-X machines, where LaTeX is harder to install and configure.

There are also other options for rendering math expressions in Sphinx, such as mathpng.py[3], which uses LaTeX to perform the rendering. There are plans to add two new math extensions to Sphinx itself in a future version: one will use jsmath [Cer07] to render math using JavaScript in the browser, and the other will use LaTeX and dvipng for rendering.

Syntax-highlighting of IPython sessions

Sphinx on its own only knows how to syntax-highlight the output of the standard `python` console. For matplotlib's documentation, we created a custom docutils formatting directive and pygments [Bra08b] lexer to color some of the extra features of the `ipython` console.

```
In [156]: fig = plt.figure()

In [157]: ax1 = fig.add_subplot(211)

In [158]: ax2 = fig.add_axes([0.1, 0.1, 0.7, 0.3])

In [159]: ax1
Out[159]: <matplotlib.axes.Subplot instance at 0xd54b26c>

In [160]: print fig.axes
[<matplotlib.axes.Subplot instance at 0xd54b26c>, <matplotlib.axes.Axes instance at 0xd3f0b2c>]
```

Framework

These new extensions are part of a complete turnkey framework for building Sphinx documentation geared specifically to Scientific Python applications. The framework is available as a subproject in matplotlib's source code repository[4] and can be used as a starting point for other projects using Sphinx.

This template is still in its early stages, but we hope it can grow into a project of its own. It could become a repository for the best ideas from other Sphinx-using projects and act as a sort of incubator for future features in Sphinx proper. This may include the web-based documentation editor currently being used by the Numpy project.

Future directions

intersphinx

Sphinx recently added "intersphinx" functionality, which allows one set of documentation to reference methods and classes etc. in another set. This opens up some very nice possibilities once a critical mass of Scientific Python tools standardize on Sphinx. For instance, the histogram plotting functionality in matplotlib could reference the underlying methods in Numpy, or related methods in Scipy, allowing the user to easily learn about all the options available without risk of duplicating information in multiple places.

Acknowledgments

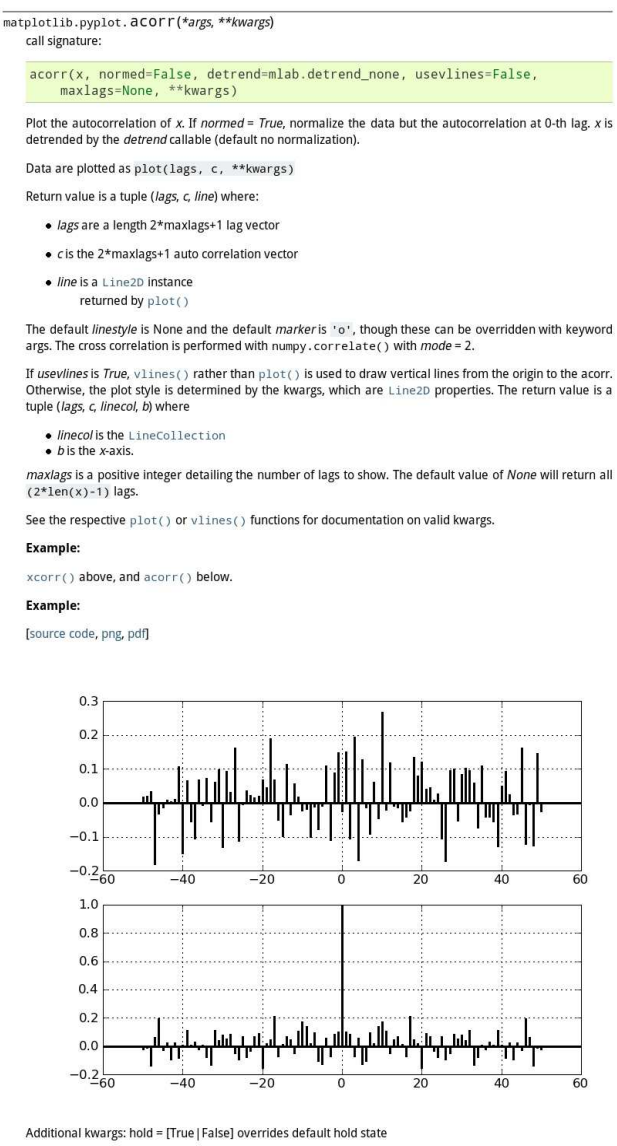
John Hunter, Darren Dale, Eric Firing and all the other matplotlib developers for their hard work on this documentation project.

[2] The license for TeX allows this, as long as we don't call it "TeX ".

[3] https://trac.fysik.dtu.dk/projects/ase/browser/trunk/doc/mathpng.py

[4] http://matplotlib.svn.sourceforge.net/viewvc/matplotlib/trunk/py4science/examples/sphinx_template/

Figures



matplotlib.pyplot.acorr(*args, **kwargs)

call signature:

```
acorr(x, normed=False, detrend=mlab.detrend_none, usevlines=False,
    maxlags=None, **kwargs)
```

Plot the autocorrelation of *x*. If *normed* = *True*, normalize the data but the autocorrelation at 0-th lag. *x* is detrended by the *detrend* callable (default no normalization).

Data are plotted as `plot(lags, c, **kwargs)`

Return value is a tuple (*lags*, *c*, *line*) where:

- *lags* are a length 2*maxlags+1 lag vector
- *c* is the 2*maxlags+1 auto correlation vector
- *line* is a `Line2D` instance returned by `plot()`

The default *linestyle* is None and the default *marker* is `'o'`, though these can be overridden with keyword args. The cross correlation is performed with `numpy.correlate()` with *mode* = 2.

If *usevlines* is *True*, `vlines()` rather than `plot()` is used to draw vertical lines from the origin to the acorr. Otherwise, the plot style is determined by the kwargs, which are `Line2D` properties. The return value is a tuple (*lags*, *c*, *linecol*, *b*) where

- *linecol* is the `LineCollection`
- *b* is the *x*-axis.

maxlags is a positive integer detailing the number of lags to show. The default value of *None* will return all `(2*len(x)-1)` lags.

See the respective `plot()` or `vlines()` functions for documentation on valid kwargs.

Example:

`xcorr()` above, and `acorr()` below.

Example:

[source code, png, pdf]

Additional kwargs: hold = [True | False] overrides default hold state

The HTML output of the `acorr` *docstring.*

References

[Bra08] G. Brandl. 2008. *Sphinx: Python Documentation Generator*. http://sphinx.pocoo.org/

[Bra08b] G. Brandl. 2008. *Pygments: Python syntax highlighter*. http://pygments.org

[Cer07] D. P. Cervone. 2007. *jsMath: A Method of Including Mathematics in Web Pages*. http://jsmath.sourceforge.net/

[Gan06] E. Gansner, E. Koustsofios, and S. North. 2006. *Drawing graphs with dot*. http://www.graphviz.org/Documentation/dotguide.pdf

[Goo06] D. Goodger. 2006. *reStructuredText: Markup Syntax and Parser Component of Docutils*. http://docutils.sourceforge.net/rst.html

[Hun08] J. Hunter, et al. 2008. *matplotlib: Python 2D plotting library*. http://matplotlib.sourceforge.net/

[Knu86] D. E. Knuth. 1986. *Computers and Typesetting, Volume B: TeX: The Program*. Reading, MA: Addison-Wesley.

[Lop08] E. Loper. 2008. *Epydoc: Automatic API Documentation Generation for Python*. http://epydoc.sourceforge.net/

[Mar01] A. Martelli. 2001. *Recipe 52305: Yet Another Python Templating Utility (YAPTU)*. From Python Cookbook. <http://code.activestate.com/recipes/52305/>

[STI08] STI Pub Companies. 2008. *STIX Font Set Project*. http://www.stixfonts.org

[Yee01] K.-P. Yee. 2001. *pydoc: Python documentation generator and online help system*. http://lfw.org/python/pydoc.html & http://www.python.org/doc/lib/module-pydoc.html.

The SciPy Documentation Project

Joseph Harrington (jh@physics.ucf.edu) – *University of Central Florida,* USA

The SciPy Documentation Project seeks to provide NumPy and SciPy with professional documentation and documentation access tools in a variety of formats. As our first effort, we have sponsored the SciPy Documentation Marathon 2008, whose goal is to produce reference documentation for the most-used portions of NumPy. I present an overview of current efforts and future plans.

Introduction and Motivation

A serious shortcoming of NumPy/SciPy is the state of its documentation. Not only does this hurt productivity and make working in NumPy very frustrating at times, it prevents adoption of the packages by many who might profitably use them. This means our pool of volunteers is much smaller than it might be, and that solving the documentation problem might thus enable projects of which today we can only dream.

Following a particularly difficult semester teaching a data analysis class in Python for the first time, I became motivated to begin the problem's resolution in the Spring 2008 semester. Below I discuss my motivating case, the requirements and design for a project to document NumPy and SciPy, our current (Summer 2008) timeline and plans, the people involved, results to date, our challenge to future development, and some issues to think about for the future.

I teach a course called Astronomical Data Analysis to upperclass physics and astronomy majors. The course initially used the Interactive Data Language, an expensive and somewhat clunky proprietary language that is popular in the astronomy community. The class is very challenging, and the focus is data analysis and how the physics of astronomical detectors affects data. Although most of the exercises are programming tasks, it is not a course in computer programming. Our use of programming is kept simple and even though about half the students typically have little programming experience, they manage to learn enough to write good and clear code (an emphasis of the course). It is perhaps a benefit that at this level the programs are kept simple and the focus is kept on analysis. The IDL course was highly effective. For example, students often reported having taught their advisors important aspects of data analysis of which the faculty advisors apparently had not been aware, and of becoming resources for their fellow students in external summer programs, graduate school, etc.

In Fall 2007, having moved institutions, I took the opportunity to switch the class to NumPy. I was pleased that most of the students in my class were comparably capable to those at my former institution. As resources for learning NumPy, they received the *Guide to NumPy*, the scipy.org web site, the mailing lists, me, source code, and the astronomy tutorial. They were also encouraged to work together. It was not enough, and the class suffered badly. Students often spent 2 hours to find and learn a simple routine. This prevented their rapid acquisition of the language, which the class depended upon. Despite their (and my) heroic efforts, none completed the final project, something that even the worst students had always been able to do. This included several students experienced in languages like C++. The problem was simple: the reference documentation was in many cases not even poor, it was nonexistent, and the other resources were insufficient to make the language accessible in the time currently expected of students and many other programmers. The conclusion was simple: I must either have documentation the next time I taught it or go back to IDL.

Requirements and Design

> If I can't find a reindeer, I'll make one instead. -T. Grinch, 1957

I was unwilling to return to IDL, but at the same time as a pre-tenure professor with a very active research program, my time was far too constrained to take on the job myself. So, I would be leading a project, not doing one. My resources were the SciPy community, some surplus research funding, and about 8 months.

The needs were similarly clear: reference pages and a reference manual for each function, class, and module; tutorial user guide; quick "Getting Started" guide; topic-specific user guides; working examples; "live" collection of worked examples (the scipy.org cookbook); tools to view and search docs interactively; variety of access methods (help(), web, PDF, etc.); mechanisms for community input and management; coverage first of NumPy, then SciPy, then other packages.

Completing the reference documentation for the routines used in the course would be sufficient to teach the class in NumPy. The requirements for the reference documentation were: a reference page per function, class, and module ("docstrings"); thorough, complete, professional text; each page accessible one level below the lowest-level expected user of the documented item; examples, cross references, mathematics, and images in the docstrings; pages for general topics (slicing, broadcasting, indexing, etc.); a glossary; a reference manual compiled from the collected pages; organized by topic areas, not alphabetically; indexed.

Timeline and Plans

I hired Stéfan van der Walt, a NumPy developer and active community member, to spearhead the writing effort and to coordinate community participation. Since it was clear that the labor needed would exceed that of a few people, we agreed that some infrastructure was needed to ensure quality documentation and even to make it possible for many non-developers to participate. This included: wiki for editing and reviewing docs, with a link to SVN, ability to generate progress statistics, etc.; templates and standards for documentation; addition of indexing, math, and image mechanisms to NumPy doc standard; production of ASCII, PDF, and HTML automatically; and workflow for editing, review, and proofing. Work on the infrastructure took about 5 weeks and was complete by June 2008.

We decided first to address NumPy, the core package needed by everyone, and to provide reference pages, a "Getting Started Guide", and a tutorial user guide. We would also need doc viewing tools beyond Python's help() function. These would initially be web browsers and PDF readers, but perhaps eventually more specialized tools would appear. We realized that we would not write the latter tool set, but that if we provided an easily parseable documentation standard, other users likely would provide them. Finally, SciPy would need the same set of documents.

Starting all projects in parallel would dilute our volunteer effort to such a degree that the UCF course would not have the documents it would need. Many efforts would likely not reach critical mass, and would fail. Also, the count of NumPy pages surprised us, as it exceeded 2300 pages. A general triage and prioritization of the needs of the course were necessary. The surviving pages of the triage are called NumPyI below:

Date	Goal
June 2008	NumPy page triage and prioritization, infrastructure
by September 2008	NumPyI 50%+ to "Needs review" status, included in release
by January 2009	NumPyI 100% to "Needs review" status, included in release
by June 2009	NumPyI Proofed, included in release
by September 2009	SciPy 25%+ to "Needs review", included in release

Results of the first stage of the project appear below. That effort raised the question of when/if/how/by whom would the "unimportant" part of NumPy be documented? It was also not clear that an effort of tens of volunteers could write a monolithic tutorial user manual. A user manual requires continuity and coherence throughout, to a much greater degree than a collection of docstrings. One possibility would be to divide the manual into around ten chapters and to allow teams to propose to write the chapters and communicate with one another to attempt to keep the document as a whole uniform and coherent.

People

The core documentation team included: Joe Harrington - organization, plans, standards and processes, funding, whip; Stéfan van der Walt (on UCF contract) - Reliable and available expert, interface with SVN, PDF and HTML document generation, standards and processes, wiki, chief writer; Emmanuelle Gouillart - wiki, wiki hosting; Pauli Virtanen - wiki, writing; Perry Greenfield - numpy.doc pages

We began the community effort with the Summer Documentation Marathon 2008, in reference to the "sprints" sponsored periodically to address particular needs of NumPy development. Volunteers immediately signed up on the documentation wiki and began writing docstrings, and Stéfan finished the UCF priority list early in the summer. As of this writing (August 2008), the following have signed up on the wiki:

Shirt	Name	Shirt	Name
y	Bjørn Ådlandsvik	y	David Huard
	René Bastian	y	Alan Jackson
	Nathan Bell	y	Teresa Jeffcott
	Joshua Bloom		Samuel John
	Patrick Bouffard	y	Robert Kern
	Matthew Brett		Roban Kramer
	Christopher Burns		Vincent Noel
y	Tim Cera	y	Travis Oliphant
	Johann Cohen-Tanugi		Scott Sinclair
	Neil Crighton	y	Bruce Southey
	Arnar Flatberg		Andrew Straw
	Pierre Gerard-Marquardt	y	Janet Swisher
y	Ralf Gommers		Theodore Test
y	Keith Goodman		James Turner
y	Perry Greenfield	y	Gael Varoquaux
y	Emmanuelle Gouillart	y	Pauli Virtanen
y	Joe Harrington		Nicky van Foreest
	Robert Hetland	y	Stéfan van der Walt
y=	earned a T-shirt! Hooray and thanks!		

During the middle of the summer, we decided to offer an incentive to attract more writers. Teresa Jeffcott at UCF produced a humorous graphic, and writers contributing over 1000 words or equivalent work in wiki

maintenance, reviewing, etc. would receive one at the SciPy 2008 conference or by mail. A number of writers signed up in response, and the offer remains good until we withdraw it. We encourage even those who simply want the shirt to sign up and write 1000 (good) words, an amount that many volunteers have contributed in a single week.

Results - Summer 2008

As of this writing, the status of the NumPy reference documentation is:

Status	%	Count
Needs editing	42	352
Being written / Changed	33	279
Needs review	20	164
Needs review (revised)	2	19
Needs work (reviewed)	0	2
Reviewed (needs proof)	0	0
Proofed	3	24
Unimportant		1531

The quality of the finished documents is easily comparable to commercial package documentation. There are many examples, they work, and the doctest mechanism ensures that they will always work. These should not be the *only* tests for a given function, as our educational tests do not necessarily exercise corner cases or even all function options. The PDF document has 309 pages. NumPyI should double that number, and it will triple from there when all documentation is complete. As of this writing, the doc wiki is the best source for NumPy documentation. It is linked from the scipy.org website under the Documentation page.

A Challenge

We would like to kill the doc problem going forward. "Normal" professional software projects, free or not, write docs and code together. *Good* software projects write docs *first* and use them as specifications. NumPy had to start fast to unite Numeric and numarray, but that era is now over (thank goodness). We thus challenge the developers to include no new functions or classes in NumPy or SciPy without documentation. We further urge those patching bugs to consider the lack of a docstring to be a major bug and to fix that problem when they fix other bugs. Those fixing bugs are particularly knowledgeable about a function, and should easily be able to write a docstring with a limited time commitment. This is particularly vital for the "unimportant" items, where there is likely to be less interest from the wider community in completing the documentation. Of course, we never wish to provide a barrier to fixing bugs. We simply state the case and ask developers to use their judgement. Likewise, if reasonable docs come with a new contribution, the details of doc standards compliance can be waived for a while, or the contribution can be accepted on a SVN branch and held until the docs pass review.

Going Forward

Since NumPy is fundamental, its reference pages come first. Then we will put SciPy on the doc wiki.

We need more writers. NumPy developers should consider addressing the "unimportant" pages, as others may lack the knowledge or motivation for doing them. The authors of specialized sections of SciPy should contribute docs for their work (initially in SVN). In some cases they are the only ones able to communicate effective use of their packages to users.

We may implement separate technical and writing reviews. It may be best to limit the reviewers to a small group, to maintain consistency and a high standard. Certainly in no case should a reviewer contribute text to a docstring, as all parts of each docstring must be seen by at least two brains.

We may need a different approach for the user guides, and we would like to start work on tools, SciPy, and user guides. For this, we need still more volunteers. So we ask you, dear reader, to go to the doc wiki, sign up, and WRITE! The time you spend will be greatly exceeded by the time you save by having docs available. Many thanks to all contributors!

Pysynphot: A Python Re-Implementation of a Legacy App in Astronomy

Victoria G. Laidler (laidler@stsci.edu) – *Computer Sciences Corporation, Space Telescope Science Institute, 3700 San Martin Drive, Baltimore, MD 21218* USA
Perry Greenfield (perry@stsci.edu) – *Space Telescope Science Institute, 3700 San Martin Drive, Baltimore, MD 21218* USA
Ivo Busko (busko@stsci.edu) – *Space Telescope Science Institute, 3700 San Martin Drive, Baltimore, MD 21218* USA
Robert Jedrzejewski (rij@stsci.edu) – *Space Telescope Science Institute, 3700 San Martin Drive, Baltimore, MD 21218* USA

Pysynphot is a package that allows astronomers to model, combine, and manipulate the spectra of stars or galaxies in simulated observations. It is being developed to replace a widely used legacy application, SYNPHOT. While retaining the data-driven philosophy of the original application, Pysynphot's architecture and improved algorithms were developed to address some of its known weaknesses. The language features available in Python and its libraries, including numpy, often enabled clean solutions to what were messy problems in the original application, and the interactive graphics capabilities of matplotlib/pylab, used with a consistent set of exposed object attributes, eliminated the need to write special-purpose plotting methods. This paper will discuss these points in some detail, as well as providing an overview of the problem domain and the object model.

Introduction

One of the things that astronomers need to do is to simulate how model stars and galaxies would look if observed through a particular telescope, camera, and filter. This is useful both for planning observations ("How long do I need to observe in order to get good signal to noise?"), and for comparing actual observations with theoretical models ("Does the real observation of this galaxy prove that my theoretical model of galaxies is correct?"). This general procedure is referred to as "synthetic photometry", because it effectively performs photometric, or brightness, measurements on synthetic (simulated) data with synthetic instruments.

This is a difficult problem to solve in generality. In addition to the intrinsic properties of a star or galaxy that determine its spectrum (the amount of light emitted as a function of wavelength), effects such as redshift and dimming by interstellar dust will also affect the spectrum when it arrives at the telescope. Real spectra are noisy with limited resolution; model spectra are smooth with potentially unlimited resolution. The response function of a telescope/instrument combination is a combination of the response of all the optical elements. And astronomers are notorious for using idiosyncratic units; in addition to the SI and cgs units, there are a variety of ways to specify flux as a function of wavelength; then there are a set of magnitude units which involve a logarithmic transformation of the flux integrated over wavelength.

A software package, SYNPHOT [Bushouse], was written in the 1980s as part of the widely-used Image Reduction and Analysis Facility, IRAF [Tody], using its proprietary language SPP. Additionally, SYNPHOT essentially has its own mini-language, in which users specify the particular combination of spectrum, bandpass, units, and functions that should be applied to construct the desired spectrum.

Motivation

As with many legacy applications, maintenance issues were a strong motivation in deciding to port to a modern language. As an old proprietary language, SPP both lacks the features of modern languages and is difficult to use with modern development tools. This raised the cost of adding new functionality; so did the rigid task-oriented architecture.

It had also become clear over the years that certain deficiencies existed in SYNPHOT at a fairly basic level:

- float arithmetic was implemented in single precision
- poor combination of wavelength tables at different resolutions
- applying redshifts sometimes lost data
- no memory caching; files used for all communications

Re-implementing SYNPHOT in Python gave us the opportunity to address these and other deficiencies.

Rather than describe Pysynphot in detail, we will provide a high-level overview of the object model, and then take a close-up look at four areas that illustrate how Python, with its object-oriented capabilities and available helper packages, made it easier for us to solve some problems, simplify others, and make a great deal of progress very quickly.

Overview of Pysynphot Objects

A Spectrum is the basic class; a Spectrum always has a wavelength table and a corresponding fluxtable in a standard internal set of units. Waveunits and Fluxunits are also associated with spectra for representation purposes. Subclasses support spectra that are created

from a file, from arrays, and from analytic functions such as a black body or a Gaussian.

A Bandpass, or SpectralElement, has a wavelength table and a corresponding dimensionless throughput table. Subclasses support bandpasses that are created from a file, from arrays, from analytic functions such as a box filter, and from a specified observing configuration of a telescope/instrument combination.

Spectrum objects can be combined with each other and with Bandpasses to produce a CompositeSpectrum.

WaveUnits and FluxUnits are created from string representations of the desired unit in the shorthand used by SYNPHOT. All the unit conversion machinery is packaged in these objects.

A Spectrum and Bandpass collaborate to produce an Observation, which is a special-purpose subclass of Spectrum used to simulate observing the Spectrum through the Bandpass.

A Spectrum also collaborates with Bandpass and Unit objects to perform renormalization ("What would this spectrum have to look like in order to have a flux of value F in units U when observed through bandpass B?").

As is evident from the above definitions, most of the objects in pysynphot have array attributes, and thus rely heavily on the array functionality provided by numpy. When Spectrum or Bandpass objects are read from or written to files, these are generally FITS files, so we also rely significantly on PyFITS. Use of these two packages is sufficiently widespread that they don't show up in any of the following closeups, but they are critical to our development effort.

There are a few other specialty classes and a number of other subclasses, but this overview is enough to illustrate the rest of the paper.

Closeup #1: Improved wavelength handling

When SYNPHOT is faced with the problem of combining spectra that are defined over different wavelength sets, it creates an array using a default grid, defined as 10000 points covering the wavelength range where the calculated passband or spectrum is non-zero. The wavelengths are spaced logarithmically over this range, such that:

log10(lambda_i) = log10(lambda_min) + (i-1)*delta

where:

- lambda_i is the ith wavelength value
- delta = (log10 (max) -log10 (min)) / (N-1)
- lambda_min = wavelength of first non-zero flux
- lambda_max = wavelength of last non-zero flux
- N = 10000

This is completely insensitive to the wavelength spacing associated with each element; the spacing in the final wavelength set is determined entirely by the total range covered. Narrow spectral features can be entirely lost when they fall entirely within one of these bins.

Pysynphot does not use a pre-determined grid like this one. Instead, a CompositeSpectrum knows about the wavelength tables of each of its components. It constructs its own wavelength table by taking the union of the points in the wavelength tables in the individual components, thus preserving the original spacing around narrow features.

Closeup #2: Units

As mentioned above, astronomers use a variety of idiosyncratic units, and need to be able to convert between them. Wavelength can be measured in microns or Angstroms (10^-8 m), or as frequency in Hz. Many of the supported fluxunits are, strictly speaking, flux densities, which represent flux per wavelength per time per area; thus the units of the flux depend on the units of the wavelength, as shown in the list below. The flux itself may be represented in photons or ergs. Magnitudes, still commonly in use by optical and infrared astronomers, are typically

$-2.5 * log_{10}(F) + ZP$

where F is the flux integrated over wavelength, and the zeropoint depends on which magnitude system you're using. To compare directly to observations, you need the flux in counts, which integrates out not only the wavelength distribution but also the effective area of the telescope's light-collecting optics.

- fnu = ergs s^-1 cm^-2 Hz^-1
- flam = ergs s^-1 cm^-2 Ang^-1
- photnu = photons s^-1 cm^-2 Hz^-1
- photlam = photons s^-1 cm^-2 Ang^-1
- jy = 10^-23 ergs s^-1 cm^-2 Hz^-1
- mjy = 10^-26 ergs s^-1 cm^-2 Hz^-1
- abmag = -2.5 log10(FNU) - 48.60
- stmag = -2.5 log10(FLAM)- 21.10
- obmag = -2.5 log10(COUNTS)
- vegamag = -2.5 log10(F/ F(VEGA))
- counts = detected counts s^-1

The SYNPHOT code includes several lengthy case statements (or the equivalent thereof), to convert from the internal units to the desired units in which a particular computation needs to be performed, and then sometimes back to the internal units.

Pysynphot's OO architecture has several benefits here. Firstly, the unit conversion code is localized within the

relevant Unit classes. Secondly, the internal representation of a spectrum always exists in the internal units (of Angstroms and Photlam), so there's never a need to convert back to it. Finally, Unit classes know whether they are flux densities, magnitudes, or counts, which simplifies the tests needed in the various conversions.

Closeup #3: From command language to OO UI

A significant portion of the SYNPHOT codebase is devoted to parsing and interpreting the mini-language in which users specify a command. These specifications can be quite long, because they can include multiple filenames of spectra to be arithmetically combined. This command language also presents a significant learning curve to new users, to whom it is not immediately obvious that the command

```
rn(z(spec(qso_template.fits),2.0),
   box(10000.0,1.0),1.00E-13,flam)
```

means

> Read a spectrum from the file qso_template.fits, then apply a redshift of z=2. Then renormalize it so that, in a 1 Angstrom box centered at 10000 Angstroms, the resulting spectrum has a flux of 1e^-13 ergs cm^-2 s^-1 A^-1.

Choosing an object-oriented user interface entirely eliminated the need for a command parser (except temporarily, for backwards compatibility). Instead of learning a command language to specify complex constructs, users work directly with the building blocks and do the construction themselves. Although this is less concise, it gives users more direct control over the process, which itself allows for easier extensibility. The learning curve is also much shallower, as the class and method names are fairly intuitive.:

```
qso=S.FileSpectrum('qso_template.fits')
qso_z=qso.redshift(2.0)
bp=S.Box(10000,1)
qso_rn=qso_z.renorm(1e-13,'flam',bp)
```

Closeup #4: Pylab gave us graphics for free

The SYNPHOT package includes several specialized tasks to provide graphics capability. These tasks have lengthy parameter lists to allow the user to specify characteristics of the plot (limits, line type, and overplotting), as well as the usual parameters with which to specify the spectrum.

The availability of matplotlib, and particularly its pylab interface, meant that we have been able to provide quite a lot of graphics capability to our users without, as yet, having to write a single line of graphics-related code. We have tuned the user interface to provide consistent attributes that are useful for plotting and annotating plots.

As the user interface develops, we will likely develop some functions to provide "standard" annotations such as labels, title, and legend. But this is functional sugar; almost all the plotting capability provided by the SYNPHOT tasks is available out of the box, and pylab provides much more versatility and control. Most of our test users are coming to pysynphot and to pylab at the same time; their reaction to the plotting capabilities has been overwhelmingly positive.

Conclusion

The need to port this legacy application became an opportunity to improve it, resulting in a re-implementation with improved architecture rather than a direct port. Python's OO features and available packages made the job much easier, and Python's ability to support functional-style programming is important in lowering the adoption barrier by astronomers.

Development and testing of pysynphot are actively ongoing, with a major milestone planned for spring 2009, when it will be used by the Exposure Time Calculators during the next observing cycle for the Hubble. We are aiming for a version 1.0 release in summer 2009.

References

[Bushouse] H. Bushouse, B. Simon, "The IRAF/STSDAS Synthetic Photometry Package", *Astronomical Data Analysis Software and Systems III*, A.S.P. Conference Series, Vol. 61, 1994

[Tody] D. Tody, "The IRAF Data Reduction and Analysis System", *Instrumentation in astronomy VI*, 1986

How the Large Synoptic Survey Telescope (LSST) is using Python

Robert Lupton (rhl@astro.princeton.edu) – *Princeton University,* USA

The Large Synoptic Survey Telescope (LSST) is a project to build an 8.4m telescope at Cerro Pachon, Chile and survey the entire sky every three days starting around 2014.

The scientific goals of the project range from characterizing the population of largish asteroids which are in orbits that could hit the Earth to understanding the nature of the dark energy that is causing the Universe's expansion to accelerate.

The application codes, which handle the images coming from the telescope and generate catalogs of astronomical sources, are being implemented in C++, exported to python using swig. The pipeline processing framework allows these python modules to be connected together to process data in a parallel environment.

The Large Synoptic Survey Telescope (LSST) is a project to build an 8.4m telescope at Cerro Pachon, Chile and survey the entire sky every three days starting around 2014.

The scientific goals of the project range from characterizing the population of largish asteroids which are in orbits that could hit the Earth to understanding the nature of the dark energy that is causing the Universe's expansion to accelerate.

The application codes, which handle the images coming from the telescope and generate catalogs of astronomical sources, are being implemented in C++, exported to python using swig. The pipeline processing framework allows these python modules to be connected together to process data in a parallel environment.

Introduction

Over the last twenty-five years Astronomy has been revolutionized by the introduction of computers and CCD detectors; the former have allowed us to employ telescope designs that permit us to build telescopes with primary mirrors of diameter 8-10m, as well as handle the enormous volumes generated by the latter; for eample, the most ambitious imaging project to date, the Sloan Digital Sky Survey (SDSS [SDSS]), has generated about 15Tby of imaging data.

There are a number of projects being planned or built to carry out surveys of the sky, but the most ambitious is the Large Synoptic Survey Telescope (LSST). This is a project to build a large telescope at Cerro Pachon, Chile and survey the entire sky every three days starting around 2014. The telescope is a novel 3-mirror design with an 8.4m diameter primary mirror and will be equipped with a 3.2Gpixel camera at prime focus. The resulting field of view will have a diameter of 3.5 degrees --- 7 times the full moon's diameter (and thus imaging an area 50 times the size of the moon with every exposure). In routine operations we expect to take an image of the sky every 15s, generating a data rate of over 800 Mby/s. In order to handle these data we will be building a complex software system running on a large cluster. The LSST project is committed to an "Open-Data, Open-Source" policy which means that all data flowing from the camera will be immediately publically available, as will all of the processing software.

The large area imaged by the LSST telescope will allow us to image the entire sky every 3 (clear!) nights. This survey will be carried out through a set of 6 filters (ultra-violet, green, red, very-near-infrared, nearish-infrared, near-infrared; 320nm -- 1050nm) allowing us to characterize the spectral properties of the several billion sources that we expect to detect --- approximately equal numbers of stars and galaxies. This unprecedented time coverage (at the faint levels reached by such a large telescope, even in as short an exposure as 15s) will allow us to detect objects that move as well as those that vary their brightness. Taking the set of images at a given point, taken over the 10-year lifetime of the project, will enable us to study the extremely faint Universe over half the sky in great detail. It is perhaps worth pointing out that the Hubble Space Telescope, while able to reach very faint levels, has a tiny field of view, so it is entirely impractical to dream of using it[2] to carry out such survey projects.

The LSST's scientific goals range from studies of the Earth's immediate neighbourhood to the furthest reaches of the visible Universe.

The sensitivity to objects that move will allow us to measure the orbits of most[3] asteroids in orbits that could hit the Earth.[4] If it's any consolation, only objects larger than c. 1km are expected to cause global catastrophes, while the main threat from smaller objects is Tsunamis (the reindeer-killing object that hit Tunguska in 1908 is thought to have been c. 100m in diameter). More distant moving objects are interesting too; LSST should be able to characterise moving objects in our Galaxy at distances of thousands of light years.

The LSST's frequent measurement of the brightness of enormous numbers of sources opens the possibility of discovering new classes of astronomical objects; a spectacular example would be detecting the flash predicted to occur when two super-massive black holes merge in

[2] Or its planned successor, the James Webb Space Telescope

[3] 90% of objects larger than 140m

[4] There are other projects, such as Pan-STARRS on Haleakala on Maui, that are expected to identify many of these objects before LSST commences operations

a distant galaxy. Such mergers are expected to be a normal part of the evolution of galaxies, and should be detectable with space-based gravitational wave detectors such as LISA. The detection of an optical counterpart would dramatically increase how much we learn from such an event.

Finally, the LSST will provide extremely powerful ways of studying the acceleration of the Universe's expansion, which is generally interpreted as a manifestation of a "dark energy" that currently comprises 70% of the energy-density of the Universe. Two techniques that will be employed are studying distant type Ia supernovae (which can be used to measure the expansion history of the Universe) and measuring the distortions imposed on distant galaxies by the curvature of space caused by intervening matter.

The requirements that this system must meet are rather daunting. The accuracy specified for measurement of astronomical quantities such as brightnesses and positions of sources significantly exceeds the current state of the art, and must be achieved on a scale far too large for significant human intervention.

LSST's Computing Needs

Analyzing the data coming from LSST will require three classes of software: The applications, the middleware, and the databases. The applications codes process the incoming pixels, resulting in catalogues of detected sources' properties; this is the part of the processing that requires a understanding of both astronomical and computing algorithms. The middleware is responsible for marshalling the applications layer over a (large) cluster; it's responsible for tasks such as disk i/o, load balancing, fault tolerance, and provenance. Finally the astronomical catalogues, along with all relevant metadata, must be stored into databases and made available to the astronomical and general public (including schools --- outreach to students between Kindergarten and 12th grade is an important part of the LSST).

The entire system is of course reliant on a build system, and here we decided to eschew the gnu toolchain (automake, autoconf, gnumake, libtool) in favour of scons, a python-based build system that replaces make, much of the configure part of the auto* tools, and the joys of building shared libraries with libtool. We felt that scons support for multi-platform builds was sufficient, especially in this modern age of ANSI/ISO C++ and Posix 1003.1 compatibility. Additionally, we are using a pure python tool eups to manage our dependencies --- we strongly felt that we needed to support having multiple sets of dependent libraries simultaneously deployed on our machines (and indeed for a developer to be able to use one set in one shell, and a different set in another).

The Application Layer

The application codes are being implemented in C++, exported to python using swig. This is a different approach to that employed by the PyRAF group at the Space Telescope Science Institute [PyRAF] which defines all of its classes in python, making extensive use of numpy and scipy. For example, if your primary need is to be able to read images from disk, manipulate them (e.g. add them together or warp them) and then either pass them to an external program or write them back to disk, a numpy-based approach is a very good fit. In a similar way, if you wish to read a catalogue of objects into a python array, then the objects in the array --- corresponding to the entries in the catalogue --- are naturally created in python.

However, for the LSST, we rejected this solution as we felt that the natural place to create many objects was in C++. For example, given an image of the sky the first stage of processing (after correcting for the instrumental response) is detecting all of the objects present in the image. This isn't a particularly hard problem, but it is not one that's well matched to python as it involves searching through all the pixels in the image to determine connected sets --- and iteration through all the elements of an array is not an array-based extension such as numpy's strong point. On the otherhand, it's very natural in a language such as C or C++; as you detect each source you add its data structure to a list. There are many technologies available for linking python and C/C++ (ctypes, boost::python, swig, pyrex, cython, ...) with various strengths and weaknesses. We chose SWIG because of its non-invasive nature (when it works it simply works --- you pass it a .h file and out pops the python interface), it's level of support, and its high-level semantics --- a C++ std::list<...> becomes a python list; a C++ std::map<string, ...> becomes a python dictionary.

Where we are using Python

As described, our fundamental operations are implemented in terms of C++ classes, and in C++. Where does python fit in? The first place is in writing tests; we have used unittest to write the majority of our (admittedly still small) test suite. Because swig can be used to wrap low-level as well as high-level interfaces, we are able to write tests that would usually be coded directly in C++[6]. A downside of this is that the developer has to be sure that problems revealed are in the C++ code not the interface layer --- but in the long run we need to test both[7].

The next major application of python is as a high-level debugger[8] For example, an C++ object detector returns an std::list of detections; but are they correct? It's easy to write a little piece of python to plot the objects in an image display programme to see if they

[6] Using e.g. boost::test or CppUnit

[7] The classic problems are due to reference counting. E.g. if operator+= is given its usual C++ meaning of returning its argument, swig will generally get the reference counts wrong.

make sense; then to plot the ones that only satisfy a certain condition, and so on. This is making use of python's strengths as a fast-prototyping language, where we can keep the same list of objects while writing visualisation code --- if we were doing this in C++, we'd have to rerun the object detector as well as the display code at each iteration. A more long-lasting aspect of the same style of coding is the quality assurance that a project such as LSST is crucially dependent on. We shall have far too much data to dream of looking at more than a tiny fraction by eye, so extensive suites of analysis programmes will be run looking for anomolies, and such programmes are also naturally written in python.

Finally, we have pushed the C++ interfaces down to quite a low level (e.g. detect objects; measure positions; merge detections from multiple detection algorithms). The modularity desired by the middleware is higher --- more at the level of returning all objects from an image, with properly measured positions. The solution is to write the modules themselves in python, making calls to a sequence of C++ primitives.

Conclusions

We have adopted python as the interactive layer in a large image processing system, and are happy with the results. The majority of the pixel-level code is written in C++ for efficiency and type-safetly, while the pieces are glued together in python. We use python both as a development language, and as the implementation language to assemble scientific modules into complete functioning pipelines capable of processing the torrent of data expected from the LSST telescope.

Appendix

Integration with numpy

There are of course very good reasons for wanting our array classes to map seamlessly onto numpy's array classes --- having been through the numeric/numarray/numpy wars, we have no wish to start yet another schism. There are two issues here: How well our image classes map to (or at least play with) numpy's; and the extent to which our C++ function calls and methods return numpy arrays rather than pure python lists.

Let us deal with the former first. We have a templated Image class which looks much like any other; it may be described in terms of a strided array[9]. This is similar to numpy's 2-D array classes, but not identical. In the past (prior to swig 1.3.27) it was possible to create python classes that inherited from both numpy's ndarray and LSST's Image but this solution was fragile, and we understood the question of exactly who owned memory and when it could be safely deleted was only hazily. Another approach would be to make the LSST image classes inherit from ndarray --- but there we have problems with the C --- C++ barrier. It seems likely that a solution can be implemented, but it may not be clean.

The second question, that of numpy record arrays versus python lists, seems to be purely a matter of policy, and writing the proper swig typemaps. However, it does raise the question of how much one wants numpy's arrays to dominate the data structures of what is basically a python program.

References

[SDSS] http://www.sdss.org
[PyRAF] http://www.stsci.edu/resources/software_hardware/pyraf

[8] A similar approach was taken by the SDSS, but using TCL7.4 bound to C

[9] The internals are in fact currently implemented in terms of NASA's VisionWorkbench (http://ti.arc.nasa.gov/projects/visionworkbench)

Realtime Astronomical Time-series Classification and Broadcast Pipeline

Dan Starr (dstarr@astro.berkeley.edu) – *UC Berkeley,* USA
Josh Bloom (jbloom@astro.berkeley.edu) – *UC Berkeley,* USA
John Brewer (bizard@propellerheads.com) – *UC Berkeley,* USA

The Transients Classification Pipeline (TCP) is a Berkeley-led, Python based project which federates data streams from multiple surveys and observatories, classifies with machine learning and astronomer defined science priors, and broadcasts sources of interest to various science clients. This multi-node pipeline uses Python wrapped classification algorithms, some of which will be generated by training machine learning software using astronomer classified time-series data. Dozens of context and time-series based features are generated in real time for astronomical sources using a variety of Python packages and remote services.

Palomar 48 inch telescope, PTF survey detector on an older instrument (inset).

Project Overview

Astronomy is entering an era of large aperture, high throughput surveys with ever increasing observation cadences and data rates. Because of the increased time resolution, science with fast variability or of an explosive, "transient" nature is becoming a focus for several up and coming surveys. The Palomar Transients Factory (PTF) is one such project [PTF]. The PTF is a consortium of astronomers interested in high variability, "transient" science, and which has a dedicated survey telescope as well as several follow-up telescope resources.

Berkeley's Transients Classification Pipeline (TCP) is a Python based project that will provide real-time identification of transient science for the Palomar Transients Factory's survey telescope, which goes online in November 2008.

The PTF's survey instrument is a ~100 megapixel, 7.8 square-degree detector attached to Palomar Observatory's Samuel Oschin 48 inch diameter telescope. Taking 60 second exposures, this instrument produces up to 100 Gigabytes of raw data per night. Immediately following an observation's exposure and read-out from the detector, the data is uploaded to Lawrence Berkeley Laboratory for calibration and reduction, which results in tables of positions and flux measurements for objects found in each image. The TCP takes this resulting data and after several steps, identifies astronomical "sources" with interesting science classes, which it broadcasts to follow-up scheduling software. PTF affiliated telescopes are then scheduled to make follow-up observations for these sources.

The TCP is designed to incorporate not only the Palomar 48 inch instrument's data stream but other telescope data streams and static surveys as well. The software development and testing of the TCP makes use of SLOAN Digital Sky Survey's "stripe 82" dataset [SDSS] and the near-infrared PAIRITEL telescope's real-time data stream [PAIRITEL]. Prior to the commissioning of the Palomar instrument, the TCP will be tested using a historically derived data stream from the preceding Palomar Quest survey on the Palomar 48 inch telescope.

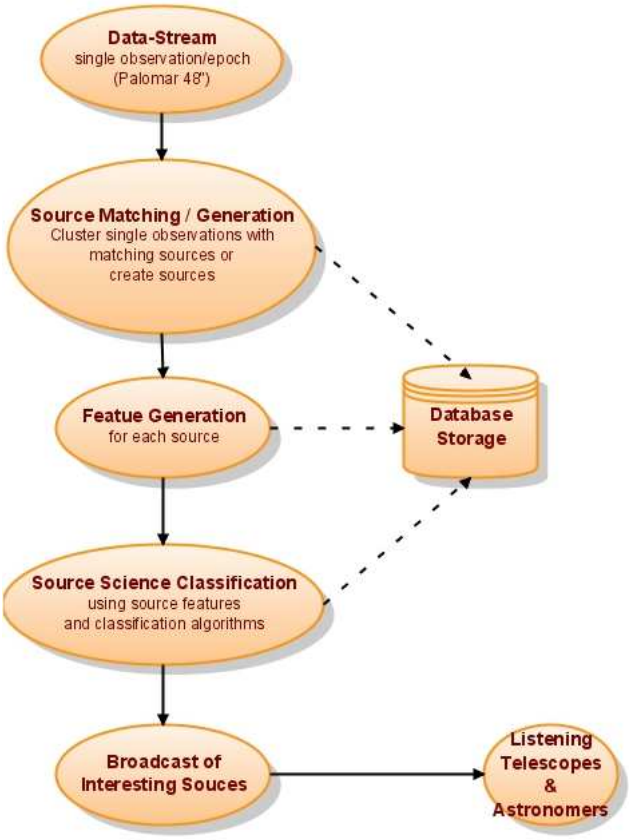
A long term goal of the Transients Classification Pipeline is to produce a scalable solution to much larger, next generation surveys such as LSST. Being a Python based project, the TCP has so far been relatively easy to implement and scale to current processing needs using the parallel nature of "IPython". Although the TCP's processing tasks are easily parallelized, care and code optimization will be needed when scaling to several orders of magnitude larger next generation survey data streams.

Two of PTF's several follow-up telescopes: Palomar 60 inch, PAIRITEL (inset).

TCP Data Flow

The TCP incorporates several data models in its design. Initially, the telescope data streams feeding into the TCP contain "object" data. Each "object" is a single flux measurement of an astronomical object at a particular time and position on the sky. The pipeline clusters these objects with existing known "sources" in the TCP's database. The clustering algorithm uses Bayesian based methods [Bloom07] and sigma cuts to make its associations. Each source then contains objects which belong to a single light source at a specific sky position, but are sampled over time. Objects not spatially associated with any sources in the TCP database are then used to define new sources.

Once a source has been created or updated with additional object data points, the TCP then generates a set of real-number "features" or properties for that source. These features can describe context information, intrinsic information such as color, or properties characterizing the source's time-series "light-curve" data. The source's features are then used by classification algorithms to determine the most probable science classes a source may belong to.



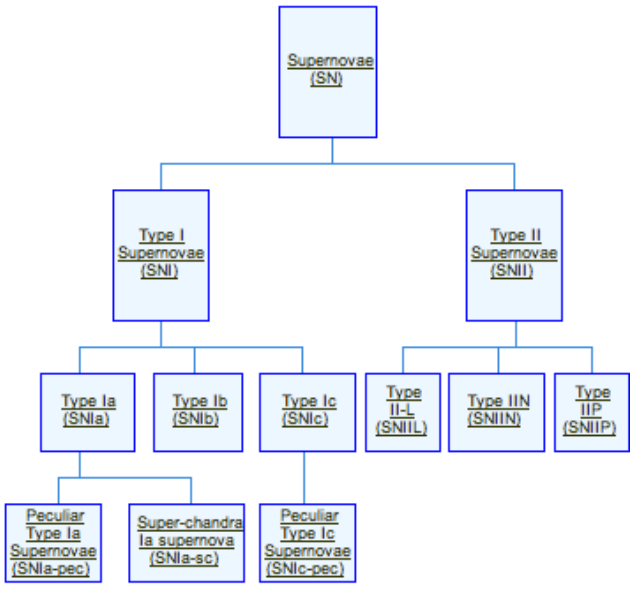
Data flow for the TCP.

Sources which match with a high probability science classes that interest PTF collaborators, will be broadcast to the PTF's "Followup Marshal" software. The Followup Marshal delegates and schedules follow-up observations on various telescopes.

One important design point of the Transients Classification Pipeline is that it allows the addition of new data streams, feature generators, and science classification algorithms while retaining its existing populated database. To meet this constraint, the TCP will use autonomous source re-evaluation software, which traverses the existing source database and re-generates features and science classifications for stale sources.

Science Classification and Follow-up

Time-variable astronomical science can be described as a hierarchy of science classes. At the top level, there are major branches such as "eruptive variables" or "eclipsing binary systems", each of which contain further refined sub-classes and child branches. Each science class has an associated set of constraints or algorithms which characterize that class ("science priors"). Determining the science classes a source belongs to can be thought of as a traverse along the class hierarchy tree; following the path where a source's characteristics ("features") fall within a science class's constraints. These science priors / classification algorithms can be either astronomer defined or generated by machine learning software which is trained using existing classified source datasets.



12 of the ~150 science classes in the current TUTOR light-curve repository.

In the case of science types which are of particular interest to the PTF group, science priors can be explicitly defined by experts in each field. This is important since PTF's primary focus is in newly appearing, sparsely sampled sources which are tricky to identify using only a couple data points. As the consortium develops a better understanding of the science coming from the Palomar 48-inch data stream, these priors can be refined. The TCP will also allow different astronomers or groups to define slightly different science priors, in order to match their specific science definitions and constraints.

Besides using astronomer crafted science priors, a primary function of the TCP is its ability to automatically

generate algorithms using an archive of classified light-curves. These algorithms are then able to distinguish science classes for sources. Also, as follow-up observations are made, or as more representative light-curves are added to the internal light-curve archive, the TCP can further refine its classification algorithms.

In a future component of the TCP, a summary of the priors / algorithms which identify a source's science class may be added to an XML representation for that source. The XML is then broadcast for follow-up observations of that source. The source's priors in this XML may be useful for follow-up in cases where the TCP was unable to decide between several science classes for that source. Here, software could parse the priors and identify any features which, if better sampled, would be useful in discerning between these unresolved science classes for that source.

The PTF's "Followup Marshal" will receive TCP's broadcasted source XMLs and then schedule follow-up observations on available telescopes. In the case of a source with unresolved science classes, the Followup Marshal may be responsible for choosing a follow-up telescope based upon the priors mentioned in the source XML.

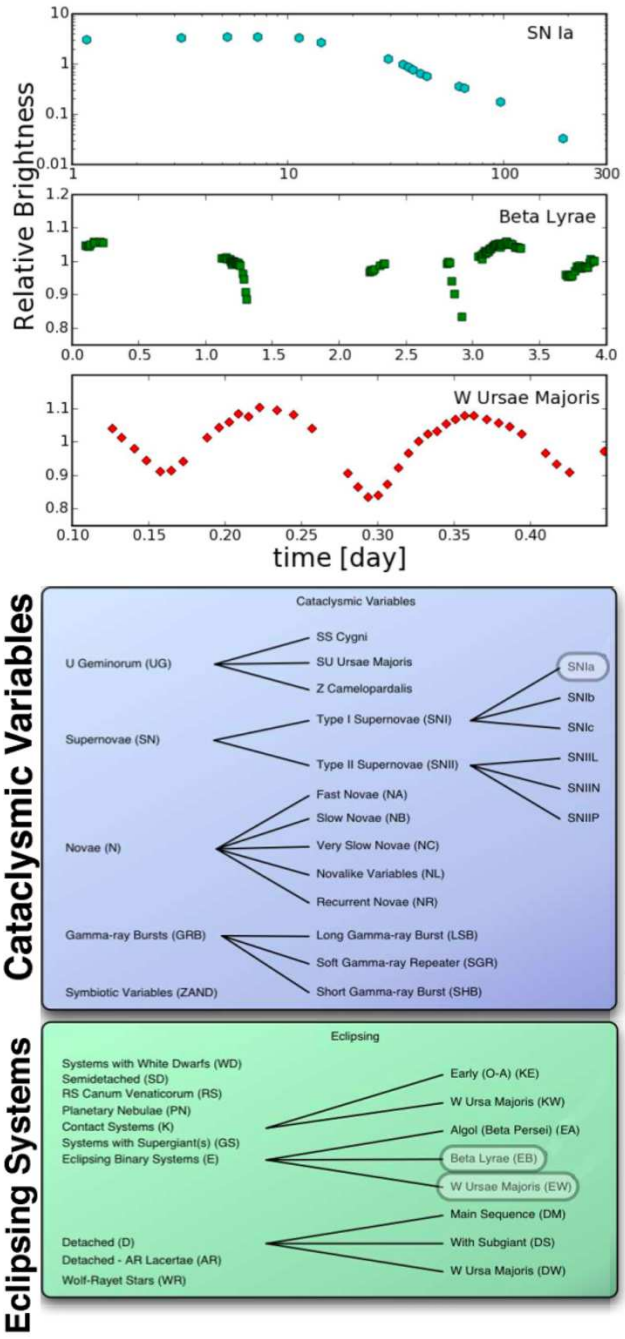
Source Features

To distinguish between a variety of different science classes, many different features need to be generated for a source. Some features represent context information, while others represent intrinsic, non-time-series stellar properties. A third feature set is derived from a source's time-series light-curve.

Context features contain source properties such as the distance of the galaxy nearest to a source, which could represent whether the source resides within a galaxy. In the case that the source closely neighbors a galaxy, the line-of-sight distance to that galaxy would also be a useful context feature of that source. The galactic latitude of a source is another context feature which roughly correlates to whether or not that source is within our galaxy. Some of these features require retrieval of information from external repositories.

Intrinsic features are neither context related or derived from time varying light-curves. The color of a stellar source is one property which astronomers tend to consider as static and unvarying over observable time.

As for time-series derived features, the TCP uses re-sampling software to make better use of locally archived example light-curves in order to generate classification algorithms applicable to the data stream instruments.



Time-series light-curves (above) and their corresponding science classes (below).

Light-Curve Resampling

Currently the TCP has an internal repository of light-curves which we've named TUTOR. As of August 2008, TUTOR contains 15000 light-curves representing 150 science classes, and derived from 87 papers and surveys. Since different instruments and telescopes were used to build the light-curves in this repository, one aspect of the TCP is to re-sample this data to better represent the observation cadences and instrument capabilities of TCP's incoming data streams.
Features generated from these re-sampled light curves are then used as training sets for machine learning software. The machine learning software then produces

science classification algorithms that are more applicable to the incoming data streams and which can be incorporated with existing science identification priors.

Python Use in the TCP

The TCP is developed in Python for several reasons. First, the TCP's primary task, generating features and determining the science classification of a source, is easily parallelized using Python. We've found the parallel aspect of IPython (formerly in the "IPython1" branch) performs well in current tests and should be applicable to the PTF data stream in November. Also, IPython's straightforward one-time importing of modules and calling of methods on client nodes made migration from TCP's original single-node pipeline trivial.

The TCP incorporates astronomer written algorithms in its feature generation software. Python makes for an easy language which programmers with differing backgrounds can code. The TCP makes use of classification code written in other languages, such as "R", but which are easily wrapped in Python. In the future, as we converge on standard science classification algorithms, this code may be re-implemented using more efficient numpy based algorithms.

An example where Python enabled code re-use, was the incorporation of PyEphem into TCP's minor planet correlation code. PyEphem wraps the C libraries of popular XEphem, which is an ephemeris calculating package.

Finally, Python has allowed TCP to make use of several storage and data transport methods. We make use of MySQL for our relational database storage, and have used the Python "dbxml" package's interface to a BerkeleyDB XML database for storage of XML and structured data. The TCP makes use of XMLRPC and socket communication, and smtplib may be used to broadcast interesting follow-up sources to astronomers.

Acknowledgments

Thanks to Las Cumbres Observatory, which partially funds the hardware and software development for the Transients Classification Project. Also thanks to UC Berkeley Astronomy students: Maxime Rischard, Chris Klein, Rachel Kennedy.

References

[PTF] http://www.mpa-garching.mpg.de/~grb07/Presentations/Kulkarni.pdf

[SDSS] Ivezi, Z., Smith, J. A., Miknaitis, G., et al., SDSS Standard Star Catalog for Stripe 82: Optical Photometry, AJ, 2007, 134, pp. 973.

[PAIRITEL] Bloom, J. S. and Starr, D. L., in ASP Conf. Ser. 351, ADASS XV, ed. C. Gabriel, C. Arviset, D. Ponz, E. Solano, 2005, pp. 156.

[Bloom07] Bloom, J.˜S., in HTU Proceedings 2007, Astron Nachrichten 329 No 3, 2008, pp. 284-287.

Analysis and Visualization of Multi-Scale Astrophysical Simulations Using Python and NumPy

Matthew Turk (mturk@slac.stanford.edu) – *KIPAC / SLAC / Stanford,* USA

The study the origins of cosmic structure requires large-scale computer simulations beginning with well-constrained, observationally-determined, initial conditions. We use Adaptive Mesh Refinement to conduct multi-resolution simulations spanning twelve orders of magnitude in spatial dimensions and over twenty orders of magnitude in density. These simulations must be analyzed and visualized in a manner that is fast, accurate, and reproducible. I present "yt," a cross-platform analysis toolkit written in Python. "yt" consists of a data-management layer for transporting and tracking simulation outputs, a plotting layer, a parallel analysis layer for handling mesh-based and particle-based data, as well as several interfaces. I demonstrate how the origins of cosmic structure – from the scale of clusters of galaxies down to the formation of individual stars – can be analyzed and visualized using a NumPy-based toolkit. Additionally, I discuss efforts to port this analysis code to other adaptive mesh refinement data formats, enabling direct comparison of data between research groups using different methods to simulate the same objects.

Analysis of Adaptive Mesh Refinement Data

I am a graduate student in astrophysics, studying the formation of primordial stars. These stars form from the collapse of large gas clouds, collapsing to higher densities in the core of extended star-forming regions. Astrophysical systems are inherently multi-scale, and the formation of primordial stars is the best example. Beginning with cosmological-scale perturbations in the background density of the universe, one must follow the evolution of gas parcels down to the mass scale of the moon to have any hope of resolving the inner structure and thus constrain the mass scale of these stars.

In order to do this, I utilize a code designed to insert higher-resolution elements within a fixed mesh, via a technique called adaptive mesh refinement (AMR). Enzo [ENZ] is a freely-available, open source AMR code originally written by Greg Bryan and now developed through the Laboratory for Computational Astrophysics by a multi-institution team of developers. Enzo is a patch-based multi-physics AMR/N-body hybrid code with support for radiative cooling, multi-species chemistry, radiation transport, and magneto-hydrodynamics. Enzo has been used to simulate a wide range of astrophysical phenomena, such as primordial star formation, galaxy clusters, galaxy formation, galactic star formation, black hole accretion and jets from gamma ray bursts. Enzo is able to insert up to 42 levels of refinement (by factors of two) allowing for a dynamic range between cells of up to 2^{42}. On the typical length scale of primordial star formation, this allows us to resolve gas parcels on the order of a hundred miles, thus ensuring the simulations fully resolve at all times the important hydrodynamics of the collapse.

A fundamental but missing aspect of our analysis pipeline was an integrated tool that was transparently parallelizable, easily extensible, freely distributable, and built on open source components, allowing for full inspection of the entire pipeline. My research advisor, Prof. Tom Abel of Stanford University, suggested I undertake the project of writing such a tool and approach it from the standpoint of attacking the problem of extremely deep hierarchies of grid patches.

Initially, `yt` was written to be a simple interface between AMR data and the plotting package "HippoDraw," which was written by Paul Kunz at the Stanford Linear Accelerator Center [HIP]. As time passed, however, it moved more toward a different mode of interaction, and it grew into a more fully-featured package, with limited data management, more abstract objects, and a full GUI and display layer built on wxPython [WX] and Matplotlib [MPL], respectively. Utilizing commodity Python-based packages, I present a fully-featured, adaptable and versatile means of analyzing large-scale astrophysical data. It is based primarily on the library NumPy [NPY], it is mostly written in Python, and it uses Matplotlib, and optionally PyTables and wxPython for various sub-tasks. Additionally, several pieces of core functionality have been moved out to C for fast numerical computation, and a TVTK-based [TVTK] visualization component is being developed.

Development Philosophy

From its beginning, `yt` has been exclusively free and open source software, and I have made the decision that it will never require components that are not open source and freely available. This enables it to be distributed, not be dependent on licensing servers, and to make available to the broader community the work put forth by me, the other developers, and the broader contributing community toward approachable analysis of the data. The development has been driven, and will continue to be driven, by my needs, and the needs of other developers.

Furthermore, no feature that I, or any other member of the now-budding development team, implement will be hidden from the community at large. This philosophy has served the toolkit well already; it has already

been examined and minor bugs have been found and corrected.

In addition to these commitments, I also sought the ability to produce publication-quality plots and to reduce the difficulty of multi-step operations. The user should be presented with a consistent, high-level, interface to the data, which will have the side-effect of enabling different entry points to the toolkit as a whole.

The development of `yt` takes place in a publicly accessible subversion repository with a Trac frontend [YT]. Sphinx-based documentation is available, and automatically updated from the subversion repository as it is checked in. In order to ease the process of installation, a script is included to install the entire set of dependencies along with the toolkit; furthermore, installations of the toolkit are maintained at several different supercomputing centers, and a binary version for Mac OS X is provided.

Organization

To provide maximum flexibility, as well as a conceptual separation of the different components and tasks to which components can be directed, `yt` is packaged into several sub-packages.

The analysis layer, `lagos`, provides several features beyond mere data access, including extensive analytical capabilities. At its simplest level, `lagos` is used to access the parameters and data in a given time-based output from an AMR simulation. However, on top of that, different means of addressing collections of data are provided, including from an intuitive object-oriented perspective, where objects are described by physical shapes and orientations.

The plotting layer, `raven`, has capabilities for plotting one-, two- and three-dimensional histograms of quantities, allowing for weighting and binning of those results. A set of pixelization routines have been written in C to provide a means of taking a set of variable-size pixels and constructing a uniform grid of values, suitable for fast plotting in Matplotlib - including cases where the plane is not axially perpendicular, allowing for oblique slices to be plotted and displayed with publication-quality rendering. Callbacks are available for overlaying analytic solutions, grid-patch boundaries, vectors, contours and arbitrary annotation.

Additionally, several other sub-packages exist that extend the functionality in various different ways. The `deliverator` package is a Turbogears-based [TG] image gallery that listens for SOAP-encoded information about images on the web, `fido` stores and retrieves data outputs, and `reason` is the wxPython-based [WX] GUI.

Object Design and Protocol

One of the difficulties in dealing with rectilinear adaptive mesh refinement data is the fundamental disconnect between the geometries of the grid structure and the objects described by the simulation. One does not expect galaxies to form and be shaped as rectangular prisms; as such, access to physically-meaningful structures must be provided. To that end, `yt` provides the following:

- Sphere
- Rectangular prism
- Cylinder / disk
- "Extracted" regions based on logical operations
- Topologically-connected sets of cells

Each of these regional descriptors is presented to the user as a single object, and when accessed the data is returned at the finest resolution available; all overlapping coarse grid cells are removed transparently. This was first implemented as physical structures resembling spheres were to be analyzed, followed by disk-like structures, each of which needed to be characterized and studied as a whole. By making available these intuitive and geometrically meaningful data selections, the underlying physical structures that they trace become more accessible to analysis and study.

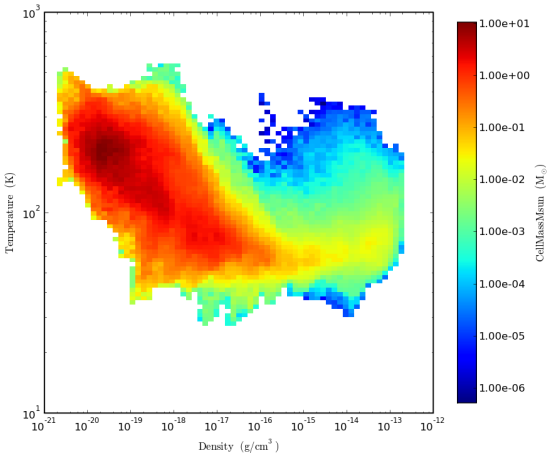
The objects are designed so that code snippets such as the following are possible:

```
>>> sp = amr_hierarchy.sphere(
...           center, radius)
>>> print sp["Density"].min()
>>> L_vec = sp.quantities["AngularMomentumVector"]()
>>> my_disk = amr_hierarchy.disk(center,
...           L_vec, radius, radius/100.0)
>>> print my_disk["Density"].min()
```

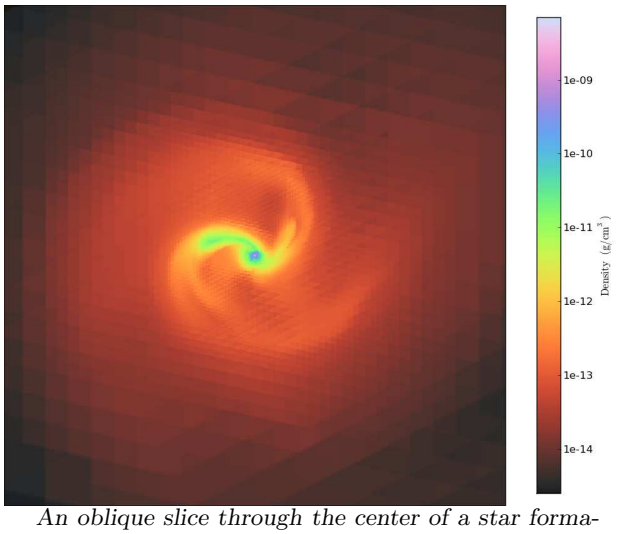
The abstraction layer is such that there are several means of interacting with these three-dimensional objects, each of which is conceptually unified, and which respects a given set of data protocols. Due to the flexibility of Python, as well as the versatility of NumPy, this functionality has been easily exposed in the form of multiple returned arrays of data, which are fast and easily manipulated. Above can be seen the calculation of the angular momentum vector of a sphere, and then the usage of that vector to construct a disk with a height relative to the radius.

These objects handle cell-based data fields natively, but are also able to appropriately select and return particles contained within them. This has facilitated the inclusion of an off-the-shelf halo finder, which allows users to quantify the clustering of particles within a region.

In addition to the object model, a flexible interface to derived data fields has been implemented. All fields, including derived fields, are allowed to be defined by either a component of a data file, or a function that transforms one or more other fields, thus allowing multiple layers of definition to exist, and allowing the user to extend the existing field set as needed. Furthermore, these fields can rely on the cells from neighboring grid patches - which will be generated automatically by `yt` as needed - which enables the creation of fields that rely on finite-difference stencils.



A two-dimensional phase diagram of the distribution of mass in the Density-Temperature plane for a collapsing gas cloud



An oblique slice through the center of a star formation simulation. The image plane is normal to the angular momentum vector.

The combination of derived fields, physically-meaningful data objects and a unique data-access protocol enables `yt` to construct essentially arbitrary representations of arbitrary collections of data. For instance, the user is able to take arbitrary profiles of data objects (radial profiles, probability distribution functions, etc) in one, two and three dimensions. These can be plotted from within the primary interface, and then output in a publication-ready format.

Two-Dimensional Data Representations

In order to make images and plots, `yt` has several different classes of two-dimensional data representations, all of which can be turned into images. Each of these objects generates a list of variable-resolution points, which are then passed into a C-based pixelization routine that transforms them into a fixed-resolution buffer, defined by a width, a height, and physical boundaries of the source data.

The simplest means of examining data is through the usage of axially-parallel slices through the dataset. This has several benefits - it is easy to calculate which grids and which cells are required to be read off disk (and most data formats allow for easy "striding" of data off disk, which reduces this operation's IO overhead) and it is easy to automate the process to step through a given dataset.

However, at some length scales in star formation problems, gas is likely to collapse into a disk, which is often not aligned with the axes of the simulation. By slicing along the axes, patterns such as spiral density waves could be missed, and ultimately go unexamined. In order to better visualize off-axis phenomena, I implemented a means of creating an image that is misaligned with the axes.

This "cutting plane" is an arbitrarily-aligned plane that transforms the intersected points into a new coordinate system such that they can be pixelized and made into a publication-quality plot. This technique required a new pixelization routine, in order to ensure that the correct voxels were taken and placed on the plot, which required an additional set of checks to determine if the voxel intersected with the image plane.

The nature of adaptive mesh refinement is such that one often wishes to examine either the sum of values along a given sight-line or a weighted-average along a given sight-line. `yt` provides an algorithm for generating line integrals in an adaptive fashion, such that every returned (x, y, v, dx, dy) point does not contain data from any points where $dx < dx_p$ or $dy < dy_p$.

We do this in a multi-step process, operating on each level of refinement in turn. Overlap between grids is calculated, such that, along the axis of projection, each grid is associated with a list of grids that it overlaps with on at least one cell. We then iterate over each level of refinement, starting with the coarsest, constructing lists of both "further-refinable" cells and "fully-refined" cells. A combination step is conducted, to combine overlapping cells from different grids; all "further-refinable" cells are passed to the next level as input to the overlap algorithm, and we continue recursing down the level hierarchy. The final projection object, with its variable-resolution cells, is returned to the user.

Once this process is completed, the projection object respects the same data protocol, and can be plotted in the same way, as an ordinary slice.

Contour Finding

Ofttimes, one needs to identify collapsing objects by finding topologically-connected sets of cells. The nature of adaptive mesh refinement, where in a given set cells may be connected across grid and refinement boundaries, requires sophisticated means for such identification.

Unfortunately, while locating topologically-connected sets inside a single-resolution grid is a straightforward but non-trivial problem in recursive programming, extending this in an efficient way to hierarchical datasets can be problematic. To that end, the algorithm implemented in `yt` checks on a grid-by-grid basis, retrieving an additional set of cells at the grid boundary. Any contour that crosses into these 'ghost zones' mandates a reconsideration of all grids that intersect with the currently considered grid. This process is expensive, as it operates recursively, but ensures that all contours are automatically joined.

Once contours are identified, they are split into individual derived objects that are returned to the user. This presents an integrated interface for generating and analyzing topologically-connected sets of related cells. In the past, `yt` has been used to conduct this form of analysis and to study fragmentation of collapsing gas clouds, specifically to examine the gravitational boundedness of these clouds and the scales at which fragmentation occurs.

Parallel Analysis

As the capabilities of supercomputers grow, the size of datasets grows as well. In order to meet these changing needs, I have been undertaking an effort to parallelize `yt` to run on multiple independent processing units. Specifically, I have been utilizing the Message Passing Interface (MPI) via the MPI4Py [MPI] module, a lightweight, NumPy-native wrapper that enables natural access to the C-based routines for interprocess communication. My goal has been to preserve at all times the API, such that the user can submit an unchanged serial script to a batch processing queue, and the toolkit will recognize it is being run in parallel and distribute tasks appropriately.

The tasks in `yt` that require parallel analysis can be divided into two different broad categories: those tasks that can act on data in an unordered, uncorrelated fashion, and those tasks that act on a decomposed image plane.

To parallelize the unordered analysis, a set of iterators have been implemented utilizing an initialize/finalize structure. Upon initialization of the iterator, it calls a method that determines which sets of data will be processed by which processors in the MPI group. The iteration proceeds as normal, and then, before the StopIteration exception is raised, it finalizes by broadcasting the final result to every processor. The unordered nature of the analysis allows the grids to be ordered such that disk access is minimized; on high-performance file systems, this results in close-to-ideal scaling of the analysis step.

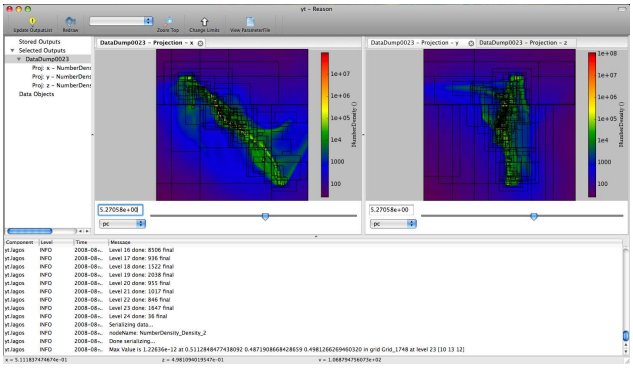
Constraints of Scale

In order to manage simulations consisting of multiple hundreds of thousands of discrete grid patches - as well as their attendant grid cell values - I have undertaken optimization using the `cProfile` module to locate and eliminate as many bottlenecks as possible. To that end, I am currently in the process of reworking the object instantiation to rely on the Python feature 'slots,' which should speed the process of generating hundreds of thousands of objects. Additionally, the practice of storing data about simulation outputs between instantiation of the Python objects has been extended; this speeds subsequent startups, and enables a more rapid response time.

Enzo data is written in one of three ways, the most efficient -and prevalent- way being via the Hierarchical Data Format (HDF5) [HDF] with a single file per processor that the simulation was run on. To limit the effect that disk access has on the process of loading data, hand-written wrappers to the HDF5 have been inserted into the code. These wrappers are lightweight, and operate on a single file at a time, loading data in the order it has been written to the disk. The package PyTables was used for some time, but the instantiation of the object hierarchy was found to be too much overhead for the brief and well-directed access desired.

Frontends and Interfaces

`yt` was originally intended to be used from the command line, and images to be viewed either in a web browser or via an X11 connection that forwarded the output of an image viewer. However, a happy side-effect of this attitude - as well as the extraordinarily versatile Matplotlib "Canvas" interface - is that the `yt` API, designed to have an a single interface to analysis tasks, is easily accessed and utilized by different interfaces. By ensuring that this API is stable and flexible, GUIs, web-interfaces, and command-line scripts can be constructed to perform common tasks.



A typical session inside the GUI

For scientific computing as a whole, such flexibility is invaluable. Not all environments have access to the same level of interactivity; for large-scale datasets, being able to interact with the data through a scripting interface enables submission to a batch processing queue, which enables appropriate allocation of resources. For smaller datasets, the process of interactively exploring datasets via graphical user interfaces,

exposing analytical techniques not available to an offline interface, is extremely worthwhile, as it can be highly immersive.

The canonical graphical user interface is written in wxPython, and presents to the user a hierarchical listing of data objects: static outputs from the simulation, as well as spatially-oriented objects derived from those outputs. The tabbed display pane shows visual representations of these objects in the form of embedded Matplotlib figures.

Recently an interface to the Matplotlib 'pylab' interface has been prepared, which enables the user to interactively generate plots that are thematically linked, and thus display an uniform spatial extent. Further enhancements to this IPython interface, via the profile system, have been targeted for the next release.

Knoboo [KBO] has been identified as a potential web-based interface, in the same style as Sage. It is a lightweight software package designed to display executed Python code in the browser but to conduct the execution on the backend. With a disjoint web-server and execution kernel model, it enables the frontend to communicate with a remote kernel server where the data and analysis packages would reside. Because of its flexibility in execution model, I have already been able to conduct analysis remotely using Knoboo as my user interface. I intend to continue working with the Knoboo developers to enhance compatibility between `yt` and Knoboo, as web-based interfaces are a powerful way to publish analysis as well as to enable collaboration on analysis tasks.

Generalization

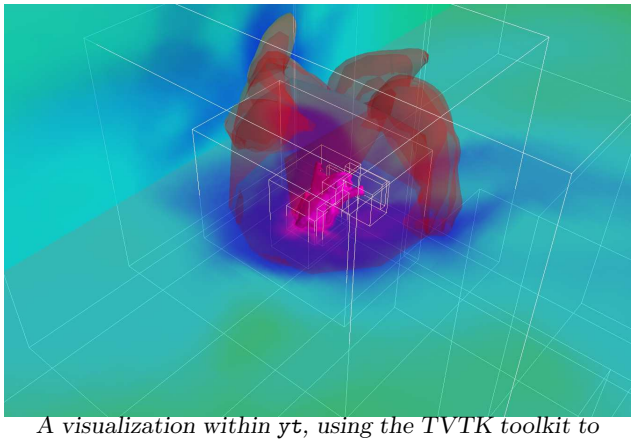
As mentioned above, `yt` was designed to handle and analyze data output from the AMR code Enzo. Dr. Jeff Oishi of Berkeley is leading the process of converting the toolkit to work equally well with data from other AMR codes; however, different codes make separate sets of assumptions about outputted data, and this must be generalized to be non-Enzo specific.

In this process, we are having to balance a desire for generalization with a desire for both simplicity and speed. To that extent, we are attempting to make minimally invasive changes where possible, and rethinking aspects of the code that were not created in the most general fashion.

By providing a unified interface to multiple, often competing, AMR codes, we will be able to utilize similar - if not identical - analysis scripts and algorithms, which will enable direct comparison of results between groups and across methods.

Future Directions

As the capabilities of `yt` expand, the ability to extend it to perform new tasks extend as well. Recently, the beginnings of a TVTK-based frontend were implemented, allowing for interactive, physically-oriented 3D visualization. This relies on the vtkCompositeDataPipeline object, which is currently weakly supported across the VTK codebase. However, the power of TVTK as an interface to VTK has significant promise, and it is a direction we are targeting.



A visualization within `yt`, using the TVTK toolkit to create 3D isocontours and cutting planes.

Work has begun on simulated observations from large-scale simulations. The first step toward this is simulating optically thin emissions, and then utilizing an analysis layer that operates on 2D image buffers.

By publishing `yt`, and generalizing it to work on multiple AMR codebases, I hope it will foster collaboration and community efforts toward understanding astrophysical problems and physical processes, while furthermore enabling reproducible research.

Acknowledgments

I'd like to acknowledge the guidance of, first and foremost, my PhD advisor Prof. Tom Abel, who inspired me to undertake this project in the first place. Additionally, I'd like to thank Jeff Oishi, a crucial member of the development team, and Britton Smith and Brian O'Shea, both of whom have been fierce advocates for the adoption of `yt` as a standard analysis toolkit. Much of this work was conducted at Stanford University and the Kavli Institute for Particle Astrophysics and Cosmology, and was supported (in part) by U.S. Department of Energy contract DE-AC02-76SF00515.

References

[ENZ] http://lca.ucsd.edu/projects/enzo
[HIP] http://www.slac.stanford.edu/grp/ek/hippodraw/
[WX] http://wxpython.org/
[MPL] http://matplotlib.sf.net/
[NPY] http://numpy.scipy.org/
[TVTK] http://svn.enthought.com/enthought/wiki/TVTK
[YT] http://yt.enzotools.org/
[TG] http://turbogears.org/
[MPI] http://mpi4py.scipy.org/
[HDF] http://hdfgroup.org/
[KBO] http://knoboo.com/

Mayavi: Making 3D Data Visualization Reusable

Prabhu Ramachandran (prabhu@aero.iitb.ac.in) – *Indian Institute of Technology Bombay, Powai, Mumbai 400076* India
Gaël Varoquaux (gael.varoquaux@normalesup.org) – *NeuroSpin, CEA Saclay, Bât 145, 91191 Gif-sur-Yvette* France

Mayavi is a general-purpose 3D scientific visualization package. We believe 3D data visualization is a difficult task and different users can benefit from an easy-to-use tool for this purpose. In this article, we focus on how Mayavi addresses the needs of different users with a common code-base, rather than describing the data visualization functionalities of Mayavi, or the visualization model exposed to the user.

Mayavi2 is the next generation of the Mayavi-1.x package which was first released in 2001. Data visualization in 3D is a difficult task; as a scientific data visualization package, Mayavi tries to address several challenges. The Visualization Toolkit [VTK] is by far the best visualization library available and we believe that the rendering and visualization algorithms developed by VTK provide the right tools for data visualization. Mayavi therefore uses VTK for its graphics. Unfortunately, VTK is not entirely easy to understand and many people are not interested in learning it since it has a steep learning curve. Mayavi strives to provide interfaces to VTK that make it easier to use, both by relying on standard numerical objects (numpy arrays) and by using the features of Python, a dynamical language, to offer simple APIs.
There are several user requirements that Mayavi strives to satisfy:

- A standalone application for visualization,
- Interactive 3D plots from IPython like those provided by pylab,
- A clean scripting layer,
- Graphical interfaces and dialogs with a focus on usability,
- Visualization engine for embedding in user dialogs box,
- An extensible application via an application framework like Envisage,
- Easy customization of the library and application,

The goal of Mayavi is to provide flexible components to satisfy *all* of these needs. We feel that there is value in reusing the core code, not only for the developers, from a software engineering point of view, but also for the users, as they can get to understand better the underlying model and concepts using the different facets of Mayavi.

Mayavi has developed in very significant ways over the last year. Specifically, every one of the above requirements have been satisfied. We first present a brief overview of the major new functionality added over the last year. The second part of the paper illustrates how we achieved the amount of reuse we have with Mayavi and what we have learned in the process of implementing this. We believe that the general ideas involved in making Mayavi reusable in these different contexts are applicable to other projects as well.

Mayavi feature overview

Starting with the Mayavi 3.0.0 release[1], there have been several significant enhancements which open up different ways of using Mayavi. We discuss each of these with examples in the following.

The `mayavi2` application

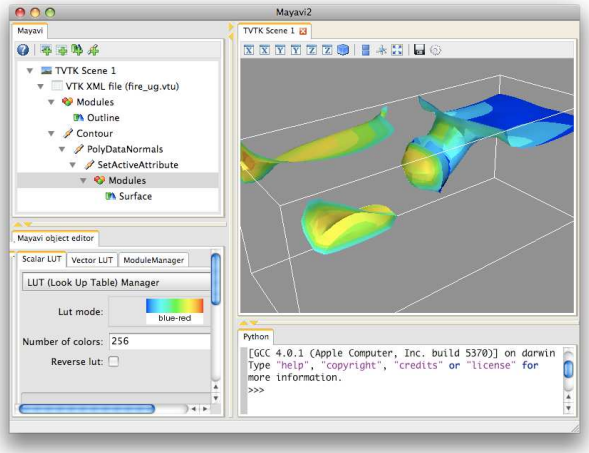
`mayavi2` is a standalone application that provides an interactive user interface to load data from files (or other sources) and visualize them interactively. It features the following:

- A powerful command line interface that lets a user build a visualization pipeline right from the command line,
- An embedded Python shell that can be used to script the application,
- The ability to drag and drop objects from the mayavi tree view on to the interpreter and script the dropped objects,
- Execution of arbitrary Python scripts in order to rapidly script the application,
- Full customization at a user level and global level. As a result, the application can be easily tailored for specific data files or workflows. For instance, the Imperial College's Applied Modeling and Computation Group has been extending Mayavi2 for triangular-mesh-specific visualizations.

[1] The name "Mayavi2" refers to the fact that the current codebase is a complete rewrite of the first implementation of Mayavi. We use it to oppose the two very different codebases and models. However the revision number of the Mayavi project is not fixed to two. The current release number is 3.0.1, although the changes between 2 and 3 are evolutionary rather than revolutionary.

- Integration into the Envisage application framework. Users may load any other plugins of their choice to extend the application. Envisage is a plugin-based application framework, similar to Eclipse, for assembling large applications from loosely-coupled components. The wing-design group at Airbus, in Bristol, designs wing meshes for simulations with a large application built with Envisage using Mayavi for the visualization.

Shown below is a visualization made on the mayavi user interface.



Screenshot of the Mayavi application.

The `mlab` interface

Mayavi's `mlab` interface provides an easy scripting interface to visualize data. It can be used in scripts, or interactively from an IPython session in a manner similar to matplotlib's pylab interface. `mlab` features the following:

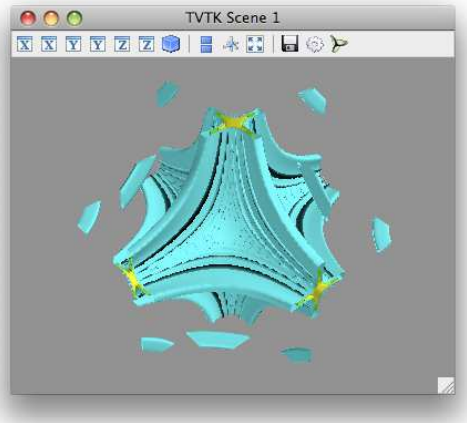
- As easy to use as possible.
- Works in the `mayavi2` application also.
- Trivial to visualize numpy arrays.
- Full power of mayavi from scripts and UI.
- Allows easy animation of data without having to recreate the visualization.

A simple example of a visualization with mlab is shown below:

```
from enthought.mayavi import mlab
from numpy import ogrid, sin

x, y, z = ogrid[-10:10:100j,
                -10:10:100j,
                -10:10:100j]

ctr = mlab.contour3d(sin(x*y*z)/(x*y*z))
mlab.show()
```



Visualization created by the above code example.

`mlab` also allows users to change the data easily. In the above example, if the scalars needs to be changed it may be easily done as follows:

```
new_scalars = x*x + y*y*0.5 + z*z*3.0
ctr.mlab_source.scalars = new_scalars
```

In the above, we use the `mlab_source` attribute to change the scalars used in the visualization. After setting the new scalars the visualization is immediately updated. This allows for powerful and simple animations.

The core features of `mlab` are all well-documented in a full reference chapter of the user-guide [M2], with examples and images.

`mlab` also exposes the lower-level mayavi API in convenient functions via the `mlab.pipeline` module. For example one could open a data file and visualize it using the following code:

```
from enthought.mayavi import mlab
src = mlab.pipeline.open('test.vtk')
o = mlab.pipeline.outline(src)
cut = mlab.pipeline.scalar_cut_plane(src)
iso = mlab.pipeline.iso_surface(src)
mlab.show()
```

`mlab` thus allows users to very rapidly script Mayavi.

Object-oriented interface

Mayavi features a simple-to-use, object-oriented interface for data visualization. The `mlab` API is built atop this interface. The central object in Mayavi visualizations is the `Engine`, which connects the different elements of the rendering pipeline. The first `mlab` example can be re-written using the `Engine` object directly as follows:

```
from numpy import ogrid, sin
from enthought.mayavi.core.engine import Engine
from enthought.mayavi.sources.api import ArraySource
from enthought.mayavi.modules.api import IsoSurface
from enthought.pyface.api import GUI

e = Engine()
e.start()
scene = e.new_scene()

x, y, z = ogrid[-10:10:100j,
                -10:10:100j,
                -10:10:100j]
data = sin(x*y*z)/(x*y*z)

src = ArraySource(scalar_data=data)
e.add_source(src)
e.add_module(IsoSurface())
GUI().start_event_loop()
```

Clearly `mlab` is a lot simpler to use. However, the raw object-oriented API of mayavi is useful in its own right, for example when using mayavi in an object-oriented context where one may desire much more explicit control of the objects and their states.

Embedding a 3D visualization

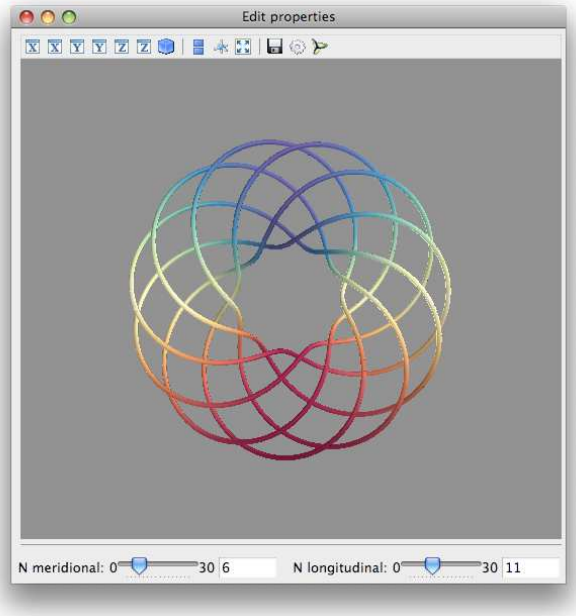
One of the most powerful features of Mayavi is the ability to embed it in a user interface dialog. One may do this either with native Traits user interfaces or in a native toolkit interface.

Embedding in TraitsUI

The TraitsUI module, used heavily throughout Mayavi to build dialogs, provides user-interfaces built on top of objects, exposing their attributes. The graphical user-interface is created in a fully descriptive way by associating object attributes with graphical editors, corresponding to views in the MVC pattern. The objects inheriting from the `HasTraits` class, the workhorse of Traits, have an embedded observer pattern, and modifying their attributes can fire callbacks, allowing the object to be manipulated live, e.g.through a GUI.

TraitsUI is used by many other projects to build graphical, interactive applications. Mayavi can easily be embedded in a TraitsUI application to be used as a visualization engine.

Mayavi provides an object, the `MlabSceneModel`, that exposes the `mlab` interface as an attribute. This object can be viewed with a `SceneEditor` in a TraitsUI dialog. This lets one use Mayavi to create dynamic visualizations in dialogs. Since we are using Traits, the core logic of the dialog is implemented in the underlying object. The `modifying_mlab_source.py` example can be found in the Mayavi examples and shows a 3D line plot parametrized by two integers. Let us go over the key elements of this example, the reader should refer to the full example for more details. The resulting UI offers slider bars to change the values of the integers, and the visualization is refreshed by the callbacks.



A Mayavi visualization embedded in a custom dialog.

The outline of the code in this example is:

```
from enthought.tvtk.pyface.scene_editor import \
                SceneEditor
from enthought.mayavi.tools.mlab_scene_model \
                import MlabSceneModel
from enthought.mayavi.core.pipeline_base \
                import PipelineBase

class MyModel(HasTraits):
    [...]

    scene = Instance(MlabSceneModel, ())
    plot = Instance(PipelineBase)

    # The view for this object.
    view = View(Item('scene',
                     editor=SceneEditor(),
                     height=500, width=500,
                     show_label=False),
                [...])

    def _plot_default(self):
        x, y, z, t = curve(self.n_merid, self.n_long)
        return self.scene.mlab.plot3d(x, y, z, t)

    @on_trait_change('n_merid,n_long')
    def update_plot(self):
        x, y, z, t = curve(self.n_merid, self.n_long)
        self.plot.mlab_source.set(x=x, y=y,
                                  z=z, scalars=t)
```

The method `update_plot` is called when the `n_merid` or `n_long` attributes are modified, for instance through the UI. The `mlab_source` attribute of the plot object is used to modify the existing 3D plot without rebuilding it.

It is to be noted that the full power of the Mayavi library is available to the user in these dialogs. This is an extremely powerful feature.

Embedding mayavi in a wxPython application

Since TraitsUI provides a wxPython backend, it is very easy to embed Mayavi in a wxPython application. The previous TraitsUI code example may be embedded in a wxPython application:

```
import wx
from mlab_model import MyModel

class MainWindow(wx.Frame):
    def __init__(self, parent, id):
        wx.Frame.__init__(self, parent, id,
                                'Mayavi in Wx')
        self.mayavi = MyModel()
        self.control = self.mayavi.edit_traits(
                        parent=self,
                        kind='subpanel').control
        self.Show(True)

app = wx.PySimpleApp()
frame = MainWindow(None, wx.ID_ANY)
app.MainLoop()
```

Thus, mayavi is easy to embed in an existing application not based on traits. Currently traits supports both wxPython and Qt as backends. Since two toolkits are already supported, it is certainly possible to support

more, although that will involve a fair amount of work.

Mayavi in envisage applications

Envisage is an application framework that allows developers to create extensible applications. These applications are created by putting together a set of plugins. Mayavi2 provides plugins to offer data visualization services in Envisage applications. The `mayavi2`

application is itself an Envisage application demonstrating the features of such an extensible application framework by assembling the Mayavi visualization engine with a Python interactive shell, logging and preference mechanisms, and a docked-window that manages layout each provided as Envisage plugins.

Customization of mayavi

Mayavi provides a convenient mechanism for users to contribute new sources, filters and modules. This may be done:

- at a global, system-wide level via a `site_mayavi.py` placed anywhere on Python's `sys.path`,
- at a local, user level by placing a `user_mayavi.py` in the users `~/.mayavi2/` directory.

In either of these, a user may register new sources, filters, or modules with Mayavi's central registry. The user may also define a `get_plugins` function that returns any plugins that the `mayavi2` application should load. Thus, the Mayavi library and application are easily customizable.

Headless usage

Mayavi also features a convenient way to create offscreen animations, so long as the user has a recent enough version of VTK (5.2 and above). This allows users to create animations of their data. Consider the following simple script:

```
n_step = 36
scene = mlab.gcf()
camera = scene.camera
da = 360.0/n_step
for i in range(n_step):
    camera.azimuth(da)
    scene.reset_zoom()
    scene.render()
    mlab.savefig('anim%02d.png' % i, size=(600,600))
```

This script rotates the camera about its azimuth and saves each such view to a new PNG file. Let this script be saved as `movie.py`. If the user has another script to create the visualization (for example consider the standard `streamline.py` example) we may run these to provide an offscreen rendering like so:

```
$ mayavi2 -x streamline.py -x movie.py -o
```

The `-o` option (or `--offscreen`) turns on the offscreen rendering. This renders the images without creating a user interface for interaction but saves the PNG images. The PNG images can be combined to create a movie using other tools.

We have reviewed the various usage patterns that Mayavi provides. We believe that this variety of use cases and entry points makes Mayavi a truly reusable 3D visualization tool. Mayavi is not domain specific and may be used in any suitable context. In the next section we discuss the secrets behind this level of reusability and the lessons we learned.

Secrets and Lessons learned

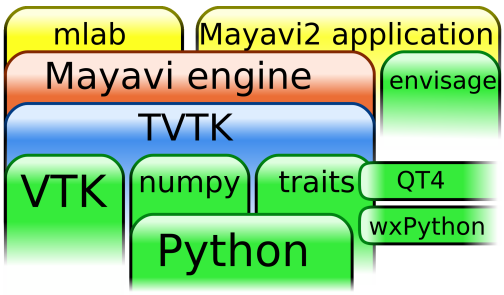
The techniques and pattern used to achieve maximal reusability in Mayavi are an application of general software architecture good practices. We will not review software architecture, although it is often underexposed to scientific developers, an introduction to the field can be found in [Gar94]. An important pattern in Mayavi's design is the separation between model and view, an introduction to which can be found in [Fow]. There are several contributing technical reasons which make Mayavi reusable:

- Layered functionality,
- Large degree of separation of UI from model,
- Object-oriented architecture and API,
- Scriptable from ground up,
- The use of Traits.

We look at these aspects in some detail in the following.

Layered functionality

Mayavi is built atop layers of functionality, and a variety of different modules:



Tool stack employed by Mayavi.

At the lowest level of this hierarchy are VTK, numpy and Traits. The TVTK package marries VTK, numpy and Traits into one coherent package. This gives us the power of VTK with a very Pythonic API. TVTK is the backbone of Mayavi. Traits optionally depends on either wxPython or Qt4 to provide a user interface.

The core Mayavi engine uses TVTK and Traits. The `mayavi2` application and the `mlab` API use the Mayavi core engine to provide data visualization. The `mayavi2` application additionally uses Envisage to provide a plugin-based extensible application.

Using Traits in the object model

The use of Traits provides us with a very significant number of advantages:

- A very powerful object model,
- Inversion of control and reactive/event-based programming: Mayavi and TVTK objects come with pre-wired callbacks which allow for easy creation of interactive applications,
- Forces a separation of UI/view from object model,
- Easy and free UIs:
  - Automatic user interfaces for wxPython and Qt4.
  - UI and scripting are well connected. This means that the UI automatically updates if the underlying model changes and this is automatically wired up with traits,
  - No need to write toolkit-specific code.

Traits allows programmers to think in very different ways and be much more efficient. It makes a significant difference to the library and allows us to completely focus on the object model.

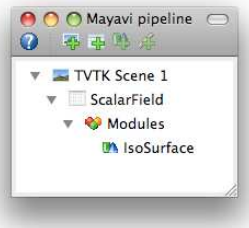
On the downsides, we note that automatically generated UIs are not very pretty. Traits provides methods to customize the UI to look better but it still isn't perfect. The layout of traits UI is also not perfect but is being improved.

Object-oriented architecture

The object-oriented API of Mayavi and its architecture helps significantly separate functionality while enabling a great deal of code reuse.

- The abstraction layers of Mayavi allows for a significant amount of flexibility and reuse. This is because the abstraction hides various details of the internals of TVTK or VTK. As an example, the Mayavi `Engine` is the object central to a Mayavi visualization that manages and encapsulates the entirety of the Mayavi visualization pipeline.
- Ability to create/extend many Mayavi engines is invaluable and is the key to much of its reusability.
- All of Mayavi's menus (on the application as well as right-click menus) are automatically generated. Similarly, the bulk of the `mlab.pipeline` interface is auto-generated. Python's ability to generate code dynamically is a big win here.
- Abstraction of menu generation based on simple metadata allows for a large degree of simplification and reuse.
- The use of Envisage for the `mayavi2` application forces us to concentrate on a reusable object model. Using envisage makes our application extensible.

The `Engine` object is not just a core object for the programming model, its functionality can also be exposed via a UI where required. This UI allows one to edit the properties of any object in the visualization pipeline as well as remove or add objects. Thus we can provide a powerful and consistent UI while minimizing duplication of efforts, both in code and design.



Dialog controlling the `Engine`: The different visualization objects are represented in the tree view. The objects can be edited by double-clicking nodes and can be added using the toolbar or via a right-click.

In summary, we believe that Mayavi is reusable because we were able to concentrate on producing a powerful object model that interfaces naturally with numpy. This is largely due to the use of Traits, TVTK and Envisage which force us to build a clean, scriptable object model that is Pythonic. The use of traits allows us to concentrate on building the object model without worrying about the view (UI). Envisage allows us to focus again on the object model without worrying too much about the need to create the application itself. We feel that, when used as a visualization engine,

Mayavi provides more than a conventional library, as it provides an extensible set of reusable dialogs that allow users to configure the visualization.

Mayavi still has room for improvement. Specifically we are looking to improve the following:

- More separation of view-related code from the object model,
- Better and more testing,
- More documentation,
- Improved persistence mechanisms,
- More polished UI.

Conclusions

Mayavi is a general-purpose, highly-reusable, 3D visualization tool. In this paper we demonstrated its reusable nature with specific examples. We also elaborated the reasons that we think make it so reusable. We believe that these general principles are capable of being applied to any project that requires the use of a user interface. There are only a few key lessons: focus on the object model, make it clean, scriptable and reusable; in addition, use test-driven development. Our technological choices (Traits, Envisage) allow us to carry out this methodology.

References

[VTK] W. Schroeder, K. Martin, W. Lorensen, "The Visualization Toolkit", Kitware, 4th edition, 2006.

[M2] P. Ramachandran, G. Varoquaux, "Mayavi2 User Guide", http://code.enthought.com/projects/mayavi/docs/development/mayavi/html/

[Gar94] D. Garlan, M. Shaw, "An Introduction to Software Architecture", in "Advances in Software Engineering and Knowledge Engineering", Volume I, eds V.Ambriola, G.Tortora, World Scientific Publishing Company, New Jersey, 1993, http://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf

[Fow] M. Fowler http://www.martinfowler.com/eaaDev/uiArchs.html

Finite Element Modeling of Contact and Impact Problems Using Python

Ryan Krauss (rkrauss@siue.edu) – *Southern Illinois University Edwardsville,* USA

This paper discusses an on going project to improve the accuracy of automotive crash simulations. Two likely causes for discrepancy between simulations and the results of real, physical tests are discussed. An existing Python package for finite element analysis, SfePy, is presented along with plans to contribute additional features in support of this work, including nonlinear material modeling and contact between two bodies.

Background and Motivation

Introduction

Automobile crashes kill 30,000 Americans in an average year [Kahane]. Motor vehicle crashes are the leading cause of death in the U.S. for the age group 4-34 [Subramanian]. Some of these deaths might be preventable. Engineers can save lives through the design of safer automobiles.

Crashworthiness design is a complicated process [Bellora] in which many important safety-related decisions must be made before realistic tests can be run. As a result, reliable virtual tests or simulations are essential, so that early design decisions can be data driven and safety countermeasures can be correctly designed.

Problem

Unfortunately, simulations are often not as accurate as they need to be. This can lead to failure of physical tests late in the design of a vehicle. Fixing such failures can be very expensive and extremely challenging. If the causes of the differences between simulation and experiment can be identified and removed, simulations could provide a true virtual test environment and these late and expensive failures could be avoided.

Two likely sources of the discrepancies between simulation and experiment include determining high speed material properties and correctly modeling contact between two bodies during simulation.

Long Term Goal

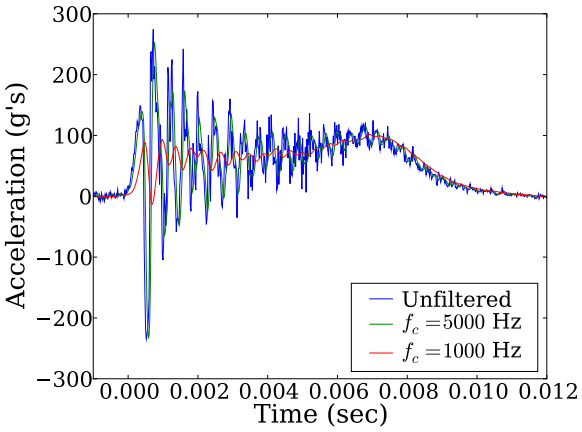
The long term goal of this research project is to remove these two obstacles to more reliable virtual tests. This will include designing a test device for impact testing of material samples. This device must be very stiff so that if its natural frequencies are excited during the impact test, the ringing can be filtered out of the data without contaminating the data itself.

The second step in achieving the long term goal of a reliable virtual test environment is creating Python based simulation software for this work, based on the SfePy [SfePy] finite element analysis package. A method for estimating material properties and generating deformation models will be developed based on the combination of simulation and experiment.

Designing a Test Device

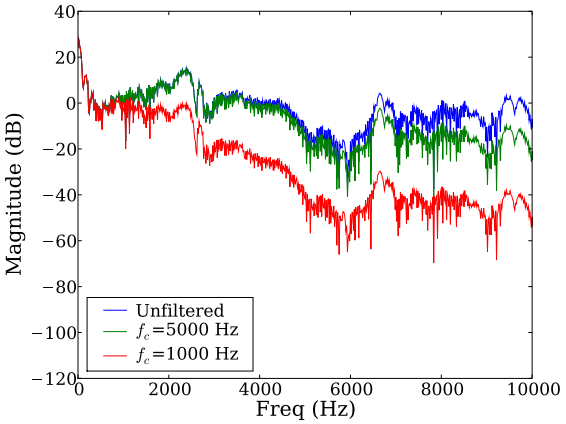
Many existing devices for high-speed materials testing contaminate the data with ringing or vibrations when used for impact testing. It is difficult to design a device with sufficiently high natural frequencies so that any ringing can be filter out without significantly altering the material response data [Maier].

The next graph shows time domain data from a device with significant ringing problems.



Impact tests results from a device with significant ringing problems

The following graph shows the fast Fourier transform (FFT) of this same data.



FFT of the data in the preceding figure

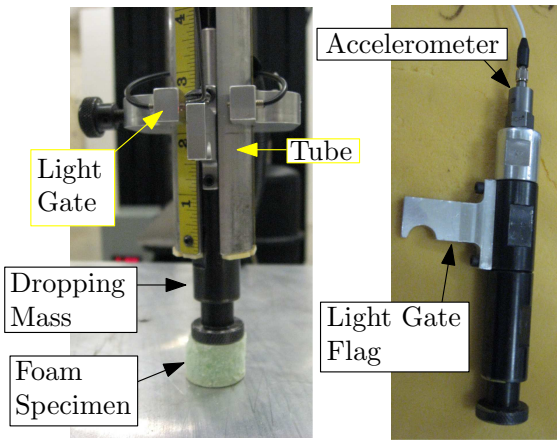
The ringing in the data shown above is not well separated in frequency from the material response data and it is difficult to filter out the ringing without altering the slope on the leading edge of the impulse.

This slope is directly related to estimates of material properties such as Young's Modulus.

Small Drop Tester

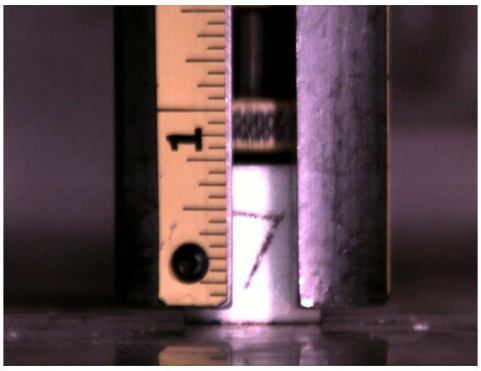
Several different impact test devices were investigated as part of this research project in the summer of 2007. The amplitude of ringing excited in a typical impact test was compared along with the separation in frequency between the ringing and the rest of the data. One device stood out: its natural frequencies were significantly higher than the rest, because it depends on a fairly small mass being dropped onto the material sample.

The device is shown in the next figure. A test consists of a small mass being dropped onto polyurethane foam samples. The mass is dropped inside a tube that is used to guide it while it is falling and to keep it vertical. The dropping mass has an accelerometer on the back of it to measure the force being exerted on the mass by the foam sample. There is also a metal flag on the side of the mass used to measure the impact velocity with a light gate.



Small mass drop tester

A picture from just before the dropping mass impacts a foam sample is shown in the next picture.

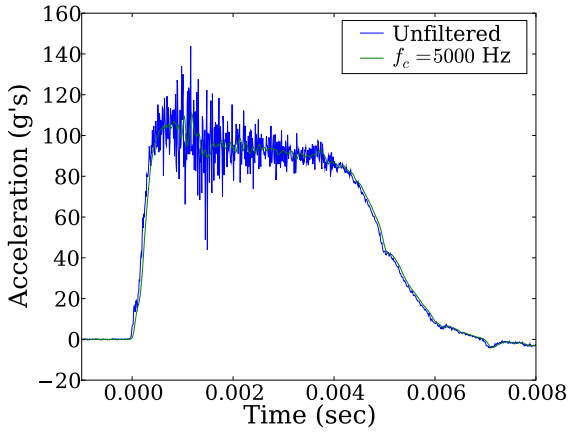


The small mass drop tester just before impact

The edges of the dropping mass were painted yellow to increase visibility in the high speed video. The white cube with the number 7 on it is the foam sample. A video clip can be seen here: http://www.siue.edu/~rkrauss/sfepy/output.avi

Example Data

An example of data from the small mass drop tester is shown below. Note that the ringing in this data is at a very high frequency and it has been largely filtered out without significantly altering the slope of the leading edge of the impulse.



Impact tests results from a device whose ringing is of a high enough frequency to be filtered without contaminating the data

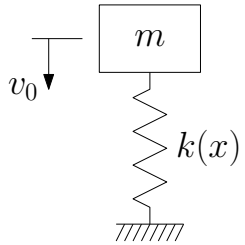
Python Based Simulations

Along with designing an impact test device that does not contaminate the data with ringing, the other goal of this project is to develop Python software modules for simulating impact tests.

The initial modeling goal is to be able to accurately simulate the small mass drop tester impacting polyurethane foam samples. This will be pursued for two reasons. First, it is necessary in order to extract the parameters for a model of the impact response of the foam. Second, it is a good candidate for an initial stepping stone toward modeling more complicated scenarios, eventually moving toward simulating in-vehicle impacts with a dummy head.

A Simple Model

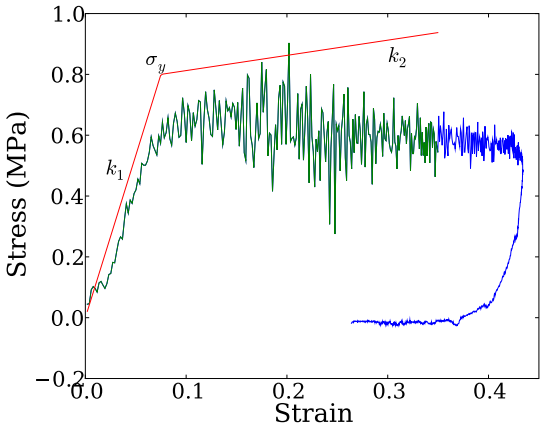
The first step in Python based simulations of the foam impact tests was to use a really simple model: a mass with an initial velocity compressing a nonlinear spring as shown below:



A simple model of the foam impact test

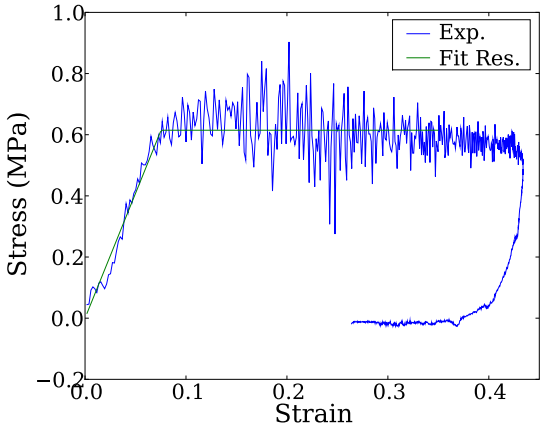
The spring was modeled as bi-linear: its force/deflection curve has a linear elastic region near

the origin and then a knee representing yielding. Above the knee, the force deflection curve is still linear, but with a different slope. The corresponding stress/strain curve can be found by dividing the force by the cross-sectional area of the foam sample and the displacement by the initial height. An example stress/strain curve is shown below. The data was curve-fit to find k_1, k_2, and σ_y using the Nelder-Mead simplex algorithm of Scipy's optimize.fmin.



Bi-linear curve fit setup

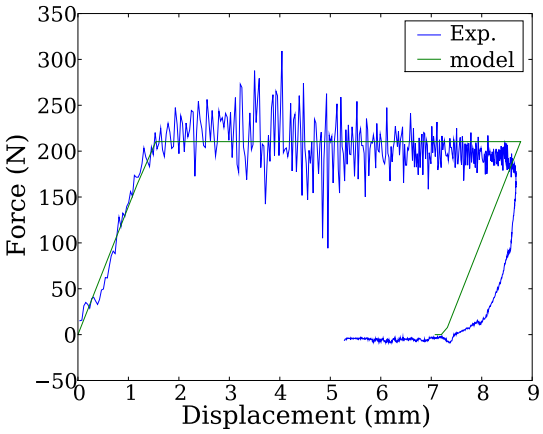
The results of the curve-fit are shown in the next graph.



Curve fit results

Model Predictions

The estimates for k_1, k_2, and σ_y were then used in an ODE model of the simple system. The resulting force vs. deflection curve is shown in the next graph. Note that the model is a fairly good fit, in spite of it being such a simple model. The primary deficiency of the model is that it does not fit the back of the force/deflection curve very well (during the rebound portion of the curve when the mass is bouncing back up).



Model predictions vs. experiment

Finite Element Modeling

While the simple model does a decent job, the accuracy of the simulation needs to be improved, especially in the rebound portion. A model is needed that accounts for the fact that the foam does not all displace uniformly. The foam is continuous and the displacement within it can vary with position (i.e. the x, y, and z coordinates of a point within the foam) as well as with time. This takes the model into the realm of partial differential equations and finite element analysis (FEA).

FEA is an approach to modeling a real, physical problem with a collection of small elements. These small elements are used to discretize a continuous problem. Ultimately, a partial differential equation model is converted to a matrix expression such as

$$[\mathbf{K}]\{\mathbf{D}\} - \{\mathbf{R}\} = 0 \qquad (1)$$

where $[\mathbf{K}]$ is the stiffness matrix, $\{\mathbf{R}\}$ is the forcing vector, and $\{\mathbf{D}\}$ is the vector of nodal displacements.

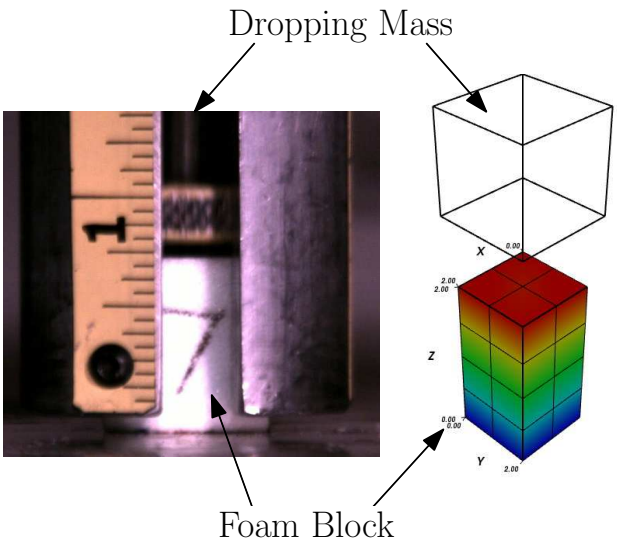
Problems with commercial FEA software

While the FEA portion of this project may be considered solved by some, the correlation between FEA simulations and the results of real, physical tests is often not as good as it should be. Most researchers are satisfied as long as key measures between simulation and experiment agree, even if the curves do not overlay closely [Rathi]. Closed-source software impedes research into the causes of these discrepancies. As such, this project seeks to add features to an existing Python FEA tool called SfePy [SfePy] so that it can fully solve this problem.

Applying FEA to this Problem

In order to apply FEA to this problem, the foam block will be discretized into many elements using a mesh. The dropping mass can be represented by one large, rigid element. The relationship between the FEA

model and the physical system is shown in the next figure.



FEA setup for this problem

Introduction to SfePy

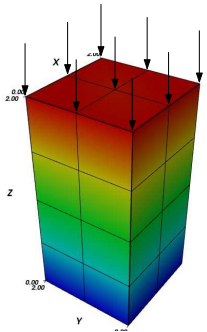
SfePy stands for Simple Finite Elements for Python. It is an FEA solver written primarily by Robert Cimrman. Along with the solver it provides a fairly high-level syntax for problem description that serves as a sort of pre-processor. SfePy also includes mesh generation capabilities for simple geometries. Post-processing must be done by some other software such as Paraview or Mayavi2.

SfePy can handle nonlinear and time varying problems and it provides solvers for linear elasticity, acoustic band gaps, Poisson's equation, and simple Navier-Stokes problems.

This project aims to contribute to SfePy capabilities for modeling nonlinear materials, multi-body dynamics, and contact between colliding bodies.

Initial FEA Model

The initial FEA model of this problem included only the foam with the interaction force between the foam and the dropping mass modeled as a traction load spread out over the top surface of the foam as shown in the next figure.



Initial FEA setup with a traction load across the top surface of the foam

Initial Input Script

Highlights of the input to SfePy are shown below. First, define the mesh file

```
filename_mesh = '3D_mesh.vtk'
```

Then label important regions of the mesh:

```
region_1 = {
    'name' : 'Bottom',
    'select' : 'nodes in (z < 0.001)'
}
region_2 = {
    'name' : 'Top',
    'select' : 'nodes in (z > 1.999)',
}
```

and use those defined regions to specify boundary conditions:

```
ebc_1 = {
    'name' : 'fixed_u',
    'region' : 'Bottom',
    'dofs' : {'u.all' : 0.0},
}
```

Material properties can be defined like this:

```
material_1 = {
    'name' : 'solid',
    'mode' : 'here',
    'region' : 'Omega',
    # Lame coefficients:
    'lame' : {'lambda' : lamb, 'mu' : mu},
}
```

Understanding FEA/SfePy

FEA is a mathematical tool for solving partial differential equations (PDEs). In the context of this problem, that means that the stress (force per unit area) and strain (change in height divided by initial height) in the foam sample are functions of more than one variable: $\sigma(x, y, z, t)$ and $\varepsilon(x, y, z, t)$.
SfePy uses the weak formulation for finite element problems. This means that the problem is stated in terms of an integral expression that is valid over a volume. For the case of solid mechanics, this integral is the potential energy functional:

$$\Pi_p = \int \{\sigma\}^T \{\mathbf{d}\varepsilon\} - \int \{\mathbf{u}\}^T \{\mathbf{\Phi}\}\, dS \qquad (2)$$

where $\{\sigma\}$ is stress, $\{\varepsilon\}$ is strain, $\{\mathbf{u}\}$ is displacement, and $\{\mathbf{\Phi}\}$ is a traction vector representing force distributed over a surface S.
The derivation in this section is based on [Cook], especially chapters 3 and 4. Discretizing and replacing the integral over the entire volume with the sum of integrals over the finite elements gives:

$$\Pi_p = \sum_{i=1}^{N_{els}} \int \int \{\sigma\}^T \{\mathbf{d}\varepsilon\}\, dV - \sum_{i=1}^{N_{els}} \int \{\mathbf{u}\}^T \{\mathbf{\Phi}\}\, dS \qquad (3)$$

Substituting a linear elastic material model

$$\{\sigma\} = [\mathbf{E}]\{\varepsilon\} \qquad (4)$$

(where $[\mathbf{E}]$ is a matrix expressing Hooke's law in three dimensions) results in

$$\int \{\sigma\}^T \{\mathbf{d}\varepsilon\} = \frac{1}{2} \{\varepsilon\}^T [\mathbf{E}] \{\varepsilon\} \tag{5}$$

and

$$\Pi_p = \sum_{i=1}^{N_{els}} \int \frac{1}{2} \{\varepsilon\}^T [\mathbf{E}] \{\varepsilon\} \, dV - \sum_{i=1}^{N_{els}} \int \{\mathbf{u}\}^T \{\mathbf{\Phi}\} \, dS \tag{6}$$

Defining the matrix $[\partial]$ for the relationship between strain ε and displacement

$$\{\varepsilon\} = [\partial] \{\mathbf{u}\} \tag{7}$$

and the matrix $[\mathbf{N}]$ for interpolation from the displacement of the corners of a brick element $\{\mathbf{d}\}$, to any point within it

$$\{\mathbf{u}\} = [\mathbf{N}] \{\mathbf{d}\} \tag{8}$$

allows the strain tensor to be written as

$$\{\varepsilon\} = [\partial] [\mathbf{N}] \{\mathbf{d}\} \text{ or } \{\varepsilon\} = [\mathbf{B}] \{\mathbf{d}\} \tag{9}$$

where $[\mathbf{B}] = [\partial] [\mathbf{N}]$.
Substituting this expression for $\{\varepsilon\}$ into equation 6 and integrating over each element produces

$$\Pi_p = \frac{1}{2} \sum_{i=1}^{N_{els}} \{\mathbf{d}\}^T [\mathbf{k}]_i \{\mathbf{d}\} - \sum_{i=1}^{N_{els}} \{\mathbf{d}\}^T \{\mathbf{r_e}\}_i \tag{10}$$

where

$$[\mathbf{k}]_i = \int [\mathbf{B}]^T [\mathbf{E}] [\mathbf{B}] \, dV \text{ and } \{\mathbf{r_e}\}_i = \int [\mathbf{N}]^T \{\mathbf{\Phi}\} \, dS \tag{11}$$

Defining a matrix $[\mathbf{L}]_i$ for each element that selects the element degrees of freedom from the global displacement vector $\{\mathbf{D}\}$

$$\{\mathbf{d}\}_i = [\mathbf{L}]_i \{\mathbf{D}\} \tag{12}$$

allows equation 10 to be rewritten as

$$\Pi_p = \frac{1}{2} \{\mathbf{D}\}^T [\mathbf{K}] \{\mathbf{D}\} - \{\mathbf{D}\}^T \{\mathbf{R}\} \tag{13}$$

where

$$[\mathbf{K}] = \sum_{i=1}^{N_{els}} [\mathbf{L}]_i^T [\mathbf{k}]_i [\mathbf{L}]_i \text{ and } \{\mathbf{R}\} = \sum_{i=1}^{N_{els}} [\mathbf{L}]_i^T \{\mathbf{r_e}\}_i \tag{14}$$

Rendering equation 13 stationary requires that

$$d\Pi_p = [\mathbf{K}] \{\mathbf{D}\} - \{\mathbf{R}\} = 0 \tag{15}$$

which is the final FEA matrix formulation that will be solved for the nodal displacement vector $\{\mathbf{D}\}$.
The interested reader is referred to [Cook] for a more thorough explanation.

References

[Kahane] Kahane, C. J., "Lives Saved by the Federal Motor Vehicle Safety Standards and Other Vehicle Safety Technologies, 1960-2002," Tech. rep., National Highway Traffic Safety Administration, Oct. 2004.

[Subramanian] Subramanian, R., "Motor Vehicle Traffic Crashes as a Leading Cause of Death in the United States, 2003," *Traffic Safety Facts, Research Note*, March 2006.

[Bellora] Bellora, V., Krauss, R., and Van Poolen, L., "Meeting interior head impact requirements: A basic scientific approach," *SAE 2001 World Congress & Exhibition, Session: Safety Test Methodology (Part C & D)*, Society of Automotive Engineers, Detroit, MI, March 2001.

[SfePy] Cimrman, R., 2008. SfePy Website. http://sfepy.kme.zcu.cz.

[Rathi] Rathi, K., Lin, T., and Mazur, D., "Evaluation of Different Countermeasures and Packaging Limits for the FMVSS201U," *SAE 2003 World Congress & Exhibition Session: Modeling of Plastic Foam and Cellular Materials for Crash Applications (Part 2 of 4)*, Society of Automotive Engineers, Detroit, MI, March 2003.

[Maier] Maier, M., Huber, U., Mkrtchyan, L., and Fremgen, C., "Recent improvements in experimental investigation and parameter fitting for cellular materials subjected to crash loads," *Composites Science and Technology*, Vol. 63, No. 14, 2003, pp. 2007-2012.

[Cook] Cook, R. D., Malkus, D. S., Plesha, M. E., and Witt, R. J., *Concepts and Applications of Finite Element Analysis*, 2002 John Wiley & Sons.

Circuitscape: A Tool for Landscape Ecology

Viral B. Shah (vshah@interactivesupercomputing.com) – *Interactive Supercomputing, Waltham, MA.* USA
Brad McRae (bmcrae@tnc.org) – *The Nature Conservancy, Seattle, WA.* USA

The modeling of ecological connectivity across networks and landscapes is an active research area that spans the disciplines of ecology, conservation, and population genetics. Recently, concepts and algorithms from electrical circuit theory have been adapted to address these problems. The approach is based on linkages between circuit and random walk theories, and has several advantages over previous analytic approaches, including incorporation of multiple dispersal pathways into analyses. Here we describe Circuitscape, a computational tool developed for modeling landscape connectivity using circuit theory. Our Python implementation can quickly solve networks with millions of nodes, or landscapes with millions of raster cells.

Introduction

Modeling of ecological connectivity across landscapes is important for understanding a wide range of ecological processes, and for achieving environmental management goals such as conserving threatened plant and animal populations, predicting infectious disease spread, and maintaining biodiversity [Cro06]. Understanding broad-scale ecological processes that depend on connectivity, and incorporating connectivity into conservation planning efforts, requires quantifying how connectivity is affected by environmental features. Thus, there is a need for efficient and reliable tools that relate landscape composition and pattern to connectivity for ecological processes.

Recently, concepts and algorithms from electrical circuit theory have been adapted for these purposes ([Mcr06], [Mcr08]). The application of circuit theory to ecological problems is motivated in part by intuitive connections between ecological and electrical connectivity: as multiple or wider conductors connecting two electrical nodes allow greater current flow than would a single, narrow conductor, multiple or wider habitat swaths connecting populations or habitats allow greater movement between them. In addition, rigorous connections between circuit and random walk theories [Doy84] mean that current, voltage, and resistance in electrical circuits all have concrete interpretations in terms of individual movement probabilities [Mcr08]. Such models can be useful for conservation planning and for predicting ecological and genetic effects of spatial heterogeneity and landscape change; for example, effective resistances calculated across landscapes have been shown to markedly improve predictions of gene flow for plant and animal species [Mcr07].

Here we describe Circuitscape, a computational tool which applies circuit-theoretic connectivity analyses to landscape data using large-scale combinatorial and numerical algorithms [Sha07].[1]

Applying circuit theory to predict landscape connectivity

In spatial ecology and conservation applications, landscapes are typically mapped as grids of raster cells in a geographical information system (GIS). For connectivity analyses, grid cells represent varying qualities of habitat, dispersal routes, or movement barriers. These raster grids can be represented as graphs, with each grid cell replaced by a node and connected to its neighbors by edges, with edge weights proportional to movement probabilities or numbers of migrants exchanged. Edges are assumed to be undirected, which implies that dispersal is balanced. Heterogeneity in landscape characteristics will typically cause movement probabilities to vary, resulting in graphs with heterogeneous edge weights.

These graphs can be analyzed using circuit theory to predict different aspects of connectivity and movement probabilities. Overviews of the theory and applications of circuit theory in ecology, conservation, and genetics are presented in [Mcr06] and [Mcr08], and will only be summarized here. Briefly, effective resistance across networks can be used as both a distance measure and a measure of redundancy in connections across graphs, and can be related to random walk times between nodes [Cha97]. Current flowing across a graph also has interpretations in terms of random walks, with current densities along circuit branches reflecting net passage probabilities for random walkers passing through nodes or across edges [Doy84]. Similarly, voltages measured in circuits can be used to predict probabilities that a random walker will reach one destination (e.g., a habitat patch) or state (e.g., death) before another [Doy84].

Computing resistance, current, and voltage with Circuitscape

Circuitscape was developed to apply circuit theory to problems in landscape ecology, which can require operations on large graphs. The computation typically starts with the raster cell map of a landscape exported from a GIS. The landscape is coded with resistance or conductance values assigned to each cell based on landscape features, such that conductance values are proportional to the relative probability of movement through each habitat type. Circuitscape converts the landscape into a graph, with every cell in the landscape represented by a node in the graph. Thus, an *mn* cell

map results in a graph with $k = mn$ nodes. Connections between neighboring cells in the landscape are represented as edges in the graph. Typically, a cell is connected to either its 4 first-order neighbors or its 8 first and second-order neighbors, although long distance connections are possible. Edge weights in the graph are functions of the per-cell conductance values, usually either the average resistance or average conductance of the two cells being connected. More sophisticated ways of computing edge weights may also be used.

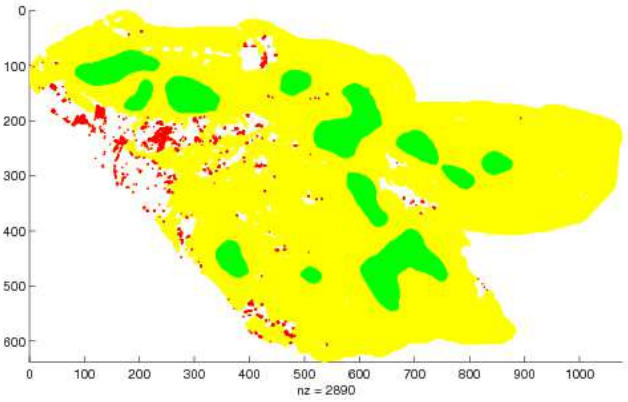
In the simplest case, we are interested in computing effective resistances, voltages, and current densities between pairs of nodes on a graph. This is done with Kirchoff's laws in matrix form. Let g_{ij} denote the conductance of the resistor connecting nodes i and j. Let G be an $n \times n$ weighted Laplacian matrix, such that $G_{ij} = -g_{ij}$ and $G_{ii} = \sum_{j=1}^{n} g_{ij}$. Resistance between nodes x and y (with $x < y$ for convenience) may be computed using a reduced conductance matrix G_y, which is the same as G but with the y^{th} row and column removed. The right hand side I is a vector with all zeros except in the x^{th} position where it is set to one. Now, solving $G_y v = I$ yields the vector v, which can then be used to derive effective resistances between nodes x and y, as well as current densities across the graph. Multiple current or voltage sources, multiple grounds, and connections to ground via resistors can be accomodated with minor modifications to this method.

The size of the graph depends on both the extent of the landscape and the resolution of the landscape data. The area modeled can vary widely across different ecological studies or conservation efforts; extents of ecological studies may be as small as a few square meters, or as large as a continent. Conservation efforts may focus on a single property, or be as large as the Yellowstone-to-Yukon project, which extends for more than 2000 miles from Yellowstone National Park to the Yukon's Mackenzie Mountains. The appropriate resolution, or cell size, for analyses will depend on kind of animal being modeled. The amount of land an animal perceives around itself typically depends on its size and tendency to move within its environment: a mountain lion may perceive about 10,000 square meters of land around it, whereas a mouse may only perceive a few square meters. Applying circuit theory to model connectivity requires working with a resolution fine enough to match the species being modeled, and an extent that may fall anywhere in the range described above. As a result, graph sizes get large very quickly. For example, a 100 km^2 area modeled for mountain lions with 100m cell sizes would yield a graph with 10,000 nodes. A landscape that includes the entire state of California would result in a graph with 40 million nodes. Landscapes that span several states can easily result in graphs with hundreds of millions of nodes; the Yellowstone-to-Yukon region includes 1.2 million km^2 of wildlife habitat, requiring 120 million nodes at 100m resolution.

Computational Methods

Circuitscape performs a series of combinatorial and numerical operations to compute a resistance-based connectivity metric. The combinatorial phase preprocesses the landscape for the subsequent numerical operations that compute resistance, current, and voltage across large graphs.

Combinatorial Methods



Combinatorial preprocessing of a landscape.

Circuitscape first reads the raster cell map from a file and constructs a graph. The raster cell map is represented by an $m \times n$ conductance matrix, where each nonzero element represents a cell of interest in the landscape. Each cell is represented by a node in the graph. Given a node in the graph, the graph construction process inserts an undirected edge connecting the node with its 4 or 8 neighbors. As a result, the graph has up to 5 or 9 nonzeros per row/column, including the diagonal. The choice of neighbor connectivity can affect connectivity of the resulting graph. A graph that is connected with 8 neighbors per cell may not be connected with 4 neighbors per cell.

The landscape graph is stored as a sparse matrix of size $mn \times mn$. This graph is extremely sparse due to the fact that every cell has at most 4 or 8 neighbors, even though the landscape may be extremely large. Edges in the graph are discovered with stencil operations. Once all the edges are discovered, the graph is converted from the triple (or co-ordinate) representation to an efficient representation such as compressed sparse rows.

A habitat patch in a landscape is represented as a collection of several neighboring nodes in the graph. Since we are typically interested in analyzing connectivity between two or more habitat patches (e.g., the patches shown in green in the figure above), all nodes in a focal habitat patch are considered collectively, and contracted into one node. Neighbors of the nodes of a habitat patch are now neighbors of the contracted node; this introduces denser rows and columns in the sparse matrix, or higher degree nodes in our graph.

Finally, we need to ensure that the graph is fully connected. Physically, there is no point in computing current flow across disconnected pieces of the

graph. Numerically, it leads to a singular system. Circuitscape ensures that the source and destination habitat patches are in the same component, and it can iterate over source destination pairs in disjoint pieces of the landscape. We prune the disconnected parts of the landscape by running a connected components algorithm on the landscape graph. In the example above, the nodes shown in red are pruned when computing current flow between the green habitat patches.

We note that the combinatorial preprocessing is not performed just once, and may need to be performed for each source/destination pair. Thus, it has to be quick even for extremely large graphs. We touch upon the software and scaling issues in a later section.

Numerical Methods

Once the graph is constructed, we form the graph Laplacian; this simply consists of making all off-diagonal entries negative and adjusting the diagonal to make the row and column sums zero. A row and column are then deleted from the graph Laplacian (making the matrix symmetric positive definite) corresponding to the destination node, indicating that it is grounded.

Effective resistance, current flows and voltages can then be computed by solving a linear system. Rows and columns corresponding to nodes connected directly to ground are deleted from the graph Laplacian, and diagonal elements corresponding to nodes connected to ground by resistors are altered by adding the conductance of each ground resistor. These modifications to the Laplacian make the matrix symmetric positive definite. The right hand side of the system is a vector of all zeros except in the position of source nodes, which are given values corresponding to the amount of current injected into each.

For small to moderate problem sizes, direct methods work well. In cases with one destination and multiple sources, the Cholesky factorization can be computed once, and then solutions can be achieved with triangular solves.

Optimal Cholesky decomposition of the 2D model problem on a unit square requires $O(n \log n)$ space and $O(n^{3/2})$ time. Although the space requirements seem modest asymptotically, they are prohibitive in practice for large problems, as shown in the table below. The number of floating point operations for the largest problems is also prohibitive. The analysis for the 2D model problem holds for matrices generated from landscapes represented as 2D grids.

Cells (10^6)	Fill (10^6)	GigaFlops
0.25	6.3	0.61
1	30	5.5
12	390	200
48	1800	1700

Time and space requirements to use sparse direct solvers.

We chose to explore iterative methods due to the large amount of memory required for Cholesky factorization, coupled with the amount of time it would take on large grids. We use preconditioned conjugate gradient to solve the linear systems with an algebraic multigrid preconditioner [Sha07].

A synthetic problem

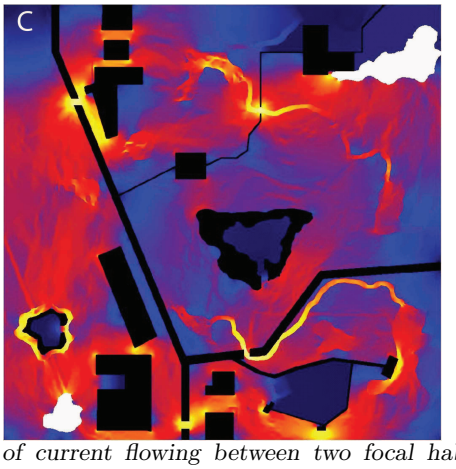
We provide some preliminary performance of Circuitscape on a synthetic problem shown below. The larger problem sizes are tiled versions of the smaller problem. While this does not represent a real application scenario, it does test the scaling of our algorithms and code on large problem sizes.

The synthetic landscape used for performance testing [Mcr08].

The largest problem we could solve had 6 million nodes. On a problem twice as large (12 million), we notice memory allocation failure (even though we believe we have sufficient memory). In the table below, we report times for the different phases of the problem.

Size	Build Graph	Components	AMG setup	Linear solve
1M	3 sec	5 sec	4 sec	9 sec
6M	14 sec	27 sec	31 sec	82 sec

Performance results for the synthetic landscape.



Map of current flowing between two focal habitat patches. Current densities through cells indicate the probability of a random walker passing each cell as it moves from one patch to the other. The map highlights portions of the landscape critical for movement between the focal patches [Mcr08].

Implementation in Python

Circuitscape was first implemented in Matlab. We decided to move to Python primarily for flexible scripting, platform independence, and potential for integration with ArcGIS (ESRI, Redlands, California, USA). The Python implementation of Circuitscape has its own GUI (designed with PythonCard), through which the user specifies the inputs and chooses from the several different problem types available.

We use numpy to manage our dense arrays, but use the sparse matrix functionality from scipy to store and manipulate the landscape graph. We use the coordinate and compressed sparse row (CSR) storage formats from scipy . We also use the conjugate gradient solver from scipy along with the algebraic multigrid preconditioner from pyamg.

One of the shortcomings of various array-based computing tools is the lack of support for operations that work with other data structures. We searched for libraries that would let us perform graph operations at scale, and were unable to find one that suited our needs. We instead wrote our own graph processing module, implementing many graph algorithms with array primitives [Sha07].

Looking forward

We are constantly trying to push the envelope on the largest problem we can solve. We are experimenting with Circuitscape on computers ranging from desktops to multiprocessor shared memory systems with 64G of RAM. We can solve problems with 6 million nodes, but larger problems appear to be restricted by memory. We are currently investigating tools that can help us identify and reduce the memory footprint of our application.

We are also working on parallelizing Circuitscape. The Matlab version of Circuitscape parallelized effortlessly with Star-P (Interactive Supercomputing, Waltham, Massachussetts, USA). For the Python implementation of Circuitscape, we are taking a different approach to parallelization. We will initially start with task-parallelism to solve several moderate sized problems simultaneously. We also plan to parallelize the linear solver, allowing users to perform connectivity computations across extremely large landscapes. Finally, we hope to integrate Circuitscape with ArcGIS to make it easy for users of ArcGIS to perform circuit-based connectivity analyses.

Circuitscape will be released under an open source license.

References

[Cha97] Chandra, A. K., P. Raghavan, W. L. Ruzzo, R. Smolensky, and P. Tiwari. 1997. The electrical resistance of a graph captures its commute and cover times. Computational Complexity 6:312–340.

[Cro06] Crooks, K. R. and Sanjayan M. (eds). 2006. Connectivity Conservation. Cambridge University Press.

[Doy84] Doyle P. G. and Snell, J. L. Random walks and electrical networks. 1984. Mathematical Association of America.

[Mcr06] McRae, B. H. 2006. Isolation by Resistance. Evolution 60:1551-1561.

[Mcr07] McRae, B. H. and Beier, P. 2007. Circuit theory predicts gene flow in plant and animal populations. Proceedings of the National Academy of Sciences of the USA 104:19885-19890.

[Mcr08] McRae, B. H., Dickson, B. G., Keitt, T. H., and Shah, V. B. In press. Using circuit theory to model connectivity in ecology and conservation. Ecology.

[Sha07] Shah, V. B. 2007. An Interactive System for Combinatorial Scientific Computing with an Emphasis on Programmer Productivity. PhD thesis, University of California, Santa Barbara.

[1]The Wilburforce Foundation sponsored this work. B. McRae is supported as a Postdoctoral Associate at the National Center for Ecological Analysis and Synthesis, a Center funded by NSF (Grant #DEB-0553768), the University of California Santa Barbara, and the State of California.

Summarizing Complexity in High Dimensional Spaces

Karl Young (karl.young@ucsf.edu) – *University of California, San Francisco,* USA

As the need to analyze high dimensional, multi-spectral data on complex physical systems becomes more common, the value of methods that glean useful summary information from the data increases. This paper describes a method that uses information theoretic based complexity estimation measures to provide diagnostic summary information from medical images. Implementation of the method would have been difficult if not impossible for a non expert programmer without access to the powerful array processing capabilities provided by SciPy.

Introduction

There is currently an explosion of data provided by high precision measurements in areas such as cosmology, astrophysics, high energy physics and medical imaging. When faced with analyzing such large amounts of high dimensional, multi-spectral data the challenge is to deduce summary information that provides physical insight into the behavior of the underlying system in a way that allows for generation and/or refinement of dynamical models.

A major issue facing those trying to analyze this type of data is the problem of dealing with a "large" number of dimensions both in the underlying index space (i.e. space or space-time) as well as the feature or spectral space of the data. Versions of the curse of dimensionality arise both from trying to generalize the methods of time series analysis to analysis in space and space-time as well as for data having a large number of attributes or features per observation.

It is here argued that information theoretic complexity measures such as those described in [Young1] can be used to generate summary information that characterizes fundamental properties of the dynamics of complex physical and biological systems. The interpretability and utility of this approach is demonstrated by the analysis of imaging studies of neurodegenerative disease in human brain. One important reason for considering such an approach is that data is often generated by a system that is non-stationary in space and/or time. This may be why statistical techniques of spatial image or pattern classification, that rely on assumptions of stationarity, have given inconsistent results when applied to magnetic resonance imaging (MRI) data. While various heuristic methods used for texture analysis have proven fruitful in particular cases of - for example - image classification, they typically do not generalize well or provide much physical insight into the dynamics of the system being analyzed. The methods described in this paper should be particularly effective in cases like classification of multi-spectral data from a particular class of physical object, i.e. for which the data to be analyzed and compared comes from a restricted class such as brain images from a set of subjects exhibiting the symptoms of one of a small class of neurodegenerative disease. The methods described allow for direct estimation of summary variables for use in classification of the behavior of physical systems, without requiring the explicit constructions described in [Crutch].

Methods

The complexity estimation methods used in this study were introduced in [Crutch] for time series analysis. The fundamental question addressed there was how much model complexity is required to optimally predict future values of a time series from past values. In addition a framework was provided for building the optimal model in the sense of being the minimally complex model required for reliable predictions.

Heuristic arguments and examples provided in [Young1] showed that only slight modifications were required to generalize the formalism for analysis of spatial data and in particular medical image data. Critical to the definition of complexity is the notion of "state" which provides the ability to predict observed values in a time series or image. Simply put, the complexity of the set of states required to describe a particular time series or image, is an indication of the complexity of the system that generated the time series or image. This in effect provides a linguistic description of the system dynamics by directly describing the structure required to infer which measurement sequences can be observed and which cannot. As described in [Crutch] to accurately and rigorously characterize the underlying complexity of a system, the set of states must in fact constitute a minimal set of optimally predictive states. How those criteria are defined and satisfied by the constructions outlined in this paper is described in [Young1], [Young2]. The simplest notion of complexity that arises from the above considerations involves a count of the number of states required for making optimal predictions. Since enumerated states can occur with varying frequencies during the course of observations, introducing the notion of state probability is natural. Shannon's original criteria for information [Shan], provides the simplest definition of an additive quantity associated with the probability distribution defined over a set of states. Complexity can then be described as an extensive quantity (i.e. a quantity that scales with measurement size) defined as the Shannon information of the probability distribution of the set of states describing the underlying system. For equally probable states this definition simply yields the log of the number of states as a measure of complexity. This notion of complexity, based on considerations of optimal prediction, is very different from the traditional

notion of Kolmogorov complexity [Cov], which quantifies series of random values as the most complex, based on considerations of incompressibility of a sequence. Here sequence is interpreted as a "program" in the context of computation, and data in the context of data analysis. Both notions of complexity provide important and complementary measures for characterizing structure in images. In the following, the optimal prediction based definition of complexity is the statistical complexity (SC) and the incompressibility based definition of complexity is entropy (H), since the Kolmogorov complexity corresponds, in the case of data analysis, to what physicists typically refer to as entropy [Cov]. A third quantity is excess entropy (EE), defined in [Feld]. EE is complementary to SC and H, and can be shown to provide important additional information. EE essentially describes the convergence rate of the entropy H to its asymptotic value as it is estimated over larger and larger volumes of the index space. The combination of EE, H and SC gives a robust characterization of the dynamics of a system.

The estimation and use of H, SC, and EE for classification of images, proceeds in 4 stages:

1. choice of an appropriate feature space (e.g. in a medical image analysis some combination of co-registered structural MRI, diffusion images,spectroscopic images, PET images, or other modalities).
2. segmentation (clustering) of feature space, i.e. clustering in the space of features without regard to coordinates (analogous to standard image segmentation for a single feature).
3. mapping of the clustered values back to the original coordinate grid and generation of estimates of H, SC, and EE from the image of clustered values.
4. classification of the data sets (e.g. images) based on the complexity estimates (e.g. via supervised or unsupervised learning algorithms)

The software implementation of the above methods is an open source package written in Python using SciPy and the Rpy [More] package to provide access to the statistical and graphical capabilities of the R statistical language [RDev] and supplemental libraries. The cluster and e1071 [Dimi] R packages were used for clustering and the AnalyzeFMRI [March] package for MR image processing. Image analysis was performed using this package on a 46 processor Beowulf cluster using the PyPAR [Niel] Python wrapper for the message passing interface (MPI). Complete (fully automated) processing of a single subject takes on the order of 40 minutes on a single 3 GHz processor.

Some important questions for future developments of the package include whether enough statistical capability will or should be provided directly in SciPy to obviate the need for inclusion of Rpy and R and how easy it will be to incorporate Ipython as a base platform for distributed processing.

In the next section I describe an illustrative analysis of structural MRI images from 23 cognitively normal (CN) subjects, 24 patients diagnosed with Alzheimer's disease (AD) and 19 patients diagnosed with frontal temporal dementia (FTD). The analysis and data are described in [Young3]. In brief: our feature space was the segmentation of each MRI image into gray matter, white matter and cerebrospinal fluid; we then applied a template of neighbouring voxels (2 neighboring voxels, compared to the next two voxels in the same line) to generate a local co-occurrence matrix of the three tissue classes, centered at each voxel; we applied the complexity metrics to this matrix, giving us an H, EE and SC measure at each voxel of each scan. We can then use regional or global summary statistics from these voxel-wise measures to classify scans according to diagnostic group.

Results

The variability of the three complexity measures in different brain regions is illustrated in Figure (1), separately for single representative CN, AD, and FTD subjects.

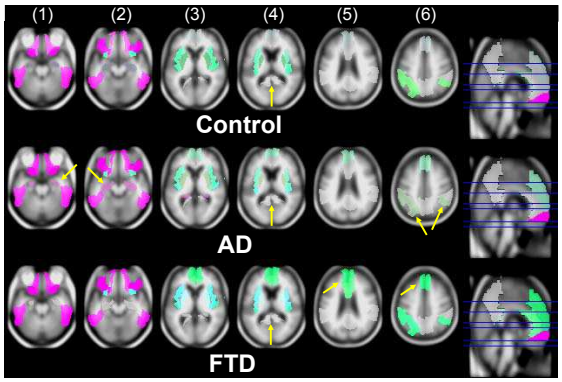


Figure 1

Simultaneous variability of entropy (H), excess entropy (EE) and statistical complexity (SC) of different brain regions in a single control subject, a single subject diagnosed with AD, and a single subject diagnosed with FTD, represented in an additive red-green-blue (RGB) color space.

An additive red-green-blue (RGB) color space is used to represent simultaneous values of H, EE, and SC. In this color space the value of H is represented on the red axis, EE on the green axis and SC on the blue axis. In this representation, a higher saturation of red represents a higher value of H, implying lack of correlation of structural patterns in an image region. Similarly, a higher saturation of green represents a higher value of EE, implying increased long range correlations of structural patterns and a higher saturation of blue represents a higher value of SC, implying an increase of locally correlated patterns. Accordingly, a simultaneous increase/decrease of all three complexity measures results in brighter/darker levels of gray. The most prominent effects in the AD subject compared to the CN and FTD subjects as seen in this representation are decreased correlation in the hippocampus (faint red regions, yellow arrows in columns 1 and 2) and diminished long range correlations of structural patterns in superior parietal lobe regions (faint green regions, arrows in column 6). In contrast, the most

prominent effect in the FTD subject compared with the CN and AD subjects is greater long range correlation in medial frontal lobe and anterior cingulum (intense green regions, arrows in columns 5 and 6).

An important practical question is whether the H, EE, and SC measures are able to distinguish between diagnostic groups as well as the current standard, which is to use local measures of cortical gray matter volume and thickness. In the following, we use logistical regression to classify scans, comparing performance using different measures.

Table (1) compares results using the structural complexity estimation against results on use of gray matter (GM) cortical thickness estimation using the FreeSurfer software on the same set of subjects.

Metric / groups	AD/CN (%)	FTD/CN (%)	AD/FTD (%)
Parietal GM volume	95 ± 4	81 ± 7	85 ± 6
Parietal GM thickness	96 ± 3	82 ± 6	86 ± 6
3 region complexity	92 ± 1	87 ± 1	91 ± 1

Table 1: logistical regression classification using Freesurfer and complexity metrics

In the table, comparisons are between classification accuracy based on structural complexity estimation and classification accuracy based on tissue volume and cortical thickness estimation (the parietal lobes provided the best separation between AD and CN subjects and the only significant separation between AD and FTD subjects for the volume and thickness estimates). For each, complexity or FreeSurfer, the regions providing the best separation between the groups are listed: for complexity the hippocampus, parietal lobe, precuneus, and Heschl's gyrus are taken together; for FreeSurfer we took measures of the thickness of parietal lobe gray matter (GM). This shows that structural complexity measures slightly outperformed volume and cortical thickness measures for the differential classification between AD and FTD as well as between FTD and CN. For the classification between AD and CN, volume and cortical thickness estimation achieved slightly higher classifications than structural complexity estimation.

Note that the classification results above may be close to the practical limit; clinician diagnosis of both AD and FTD does not have perfect agreement with post-mortem diagnosis from pathology, with errors in the same order as those reported here.

The results above compared pairwise between groups (CN vs AD, CN vs FTD, AD vs FTD). We can also assess prediction accuracy when trying to separate all three 3 groups at once, using linear discriminant analysis (LDA). This is illustrated graphically in Figures (2a), (2b), and (2c) which depict the projections onto the first two linear discriminants (labeled LD1 and LD2 in the Figures) from the LDA corresponding to the region selections for complexity estimation. This shows the expected result that group separation prominently increased with the use of focal measures, such as each of the 13 regions, as compared to global measures, such as whole brain.

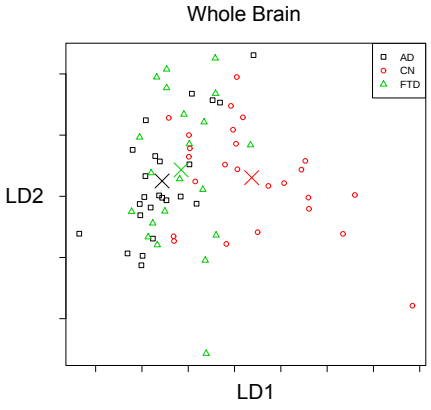


Figure 2 (a)

Results of linear discriminant analysis (LDA) using structural complexity estimates with x and y axes representing projections of complexity estimates onto the 1st and 2nd linear discriminants for the whole brain

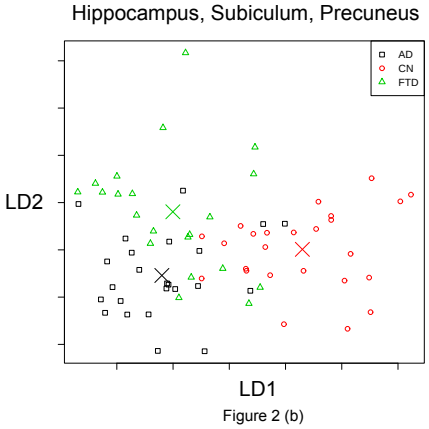


Figure 2 (b)

Results of linear discriminant analysis (LDA) using structural complexity estimates with x and y axes representing projections of complexity estimates onto the 1st and 2nd linear discriminants for the hippocampus, subiculum, and precuneus.

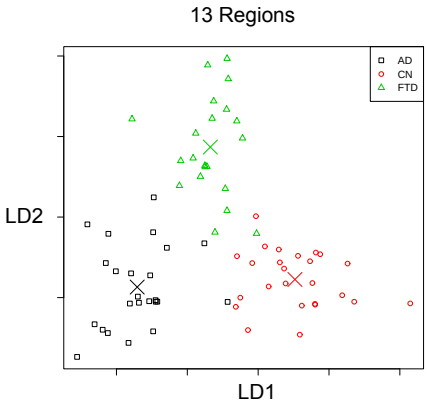


Figure 2 (c)

Results of linear discriminant analysis (LDA) using structural complexity estimates with x and y axes representing projections of complexity estimates onto the 1st and 2nd linear discriminants for all 13 regions.

Conclusion

This paper provides two main results. First, despite their simplicity and automated nature, use of structural complexity estimates is effective at capturing systematic differences on brain MRIs. They appear to be able to capture a variety of effects such as cortical volume loss and thinning. A second result is that complexity estimates can achieve similar classification separation between controls, AD and FTD patients, to that obtainable by highly specialized measures of cortical thinning. The classification accuracy provided by all of these methods is at or near the limit of the ability to reliably diagnose subjects during life, so further comparisons between methods will require improved clinical diagnosis, post-mortem diagnosis, or larger samples.

Though the complexity estimation results were promising, a number of issues remain before the methods can provide a concrete, interpretable tool suitable for clinical use. Future work will extend structural complexity estimation to multimodal imaging ([Young1]) in order to study neurodegenerative disease. This approach may be particularly effective as it does not depend on spatially confined effects in the different modalities for its classification power, as is the case for standard multivariate linear model image analysis. It also provides a more general and interpretable approach to understanding structural image properties than methods such as fractal and texture analysis.

Information theory based structural complexity estimation shows promise for use in the study and classification of large multivariate, multidimensional data sets including those encountered in imaging studies of neurodegenerative disease.

References

[Young1] Young K, Chen Y, Kornak J, Matson GB, Schuff N. Summarizing Complexity In High Dimensions. Physical Review Letters 2005; 94, 098701.

[Crutch] Crutchfield, JP, Young K. Inferring Statistical Complexity. Physical Review Letters 1989; 63, 105-107.

[Young2] Young K, Schuff N. Measuring Structural Complexity in Brain Images. Neuroimage 2008; 39(4):1721-30

[Shan] Shannon CE. A mathematical theory of communication. Bell Sys Tech Journal 1948; 27:379-423.

[Cov] Cover, T., Thomas J., 2006. Elements of Information Theory. Wiley-Interscience.

[Feld] Feldman DP, Crutchfield, JP. Structural Information in Two-Dimensional Patterns: Entropy Convergence and Excess Entropy. Physical Review E 2003; 67, 051104.

[More] Moreira, W., 2004. RPy Package. Available at http://rpy.sourceforge.net/

[RDev] R Development Core Team, 2004. R: A Language and Environment for Statistical Computing, R Foundation for Statistical Computing ISBN 3-900051-00-3. Available at http://www.r-project.org/

[Dimi] Dimitriadou, E., Hornik, K., Leisch, F., Meyer, D., Weingessel, A., 2004. R Package: e1071: Misc Functions of the Department of Statistics. Available at http://cran.r-project.org/

[March] Marchini, J.L., 2004.R Package: AnalyzeFMRI: Functions for Analysis of fMRI Datasets Stored in the ANALYZE Format. Availablae at http://cran.r-project.org/

[Niel] Nielsen O., Ciceri, G.P., Ramachandran, P., Orr, D., Kaukic, M., 2003. PyPAR - Parallel Python, Efficient and Scalable Parallelism Using the Message Passing Interface (MPI). Available at http://datamining.anu.edu.au/~ole/pypar/

[Young3] Young K, Du A, Kramer J, Rosen H, Miller B, Weiner M, Schuff N. Patterns of Structural Complexity in Alzheimer's Disease and Frontotemporal Dementia. Human Brain Mapping. In Press

Converting Python Functions to Dynamically Compiled C

Ilan Schnell (ilanschnell@gmail.com) – *Enthought*, USA

Applications written in Python often suffer from the lack of speed, compared to C and other languages which can be compiled to native machine code. In this paper we discuss ways to write functions in pure Python and still benefit from the speed provided by C code compiled to machine code. The focus is to make it as easy as possible for the programmer to write these functions.

Motivation

There are various tools (SWIG, Pyrex, Cython, boost, ctypes, an others) for creating and/or wrapping C functions into Python code. The function is either written in C or some special domain-specific language, other than Python. What these tools have in common are several inconvenience for the scientific programmer who quickly wants to accomplish a certain task:

- Learning the tool, although there are excellent references and tutorials online, the overhead (in particular for the casual programmer) is still significant.
- Dealing with additional files.
- The source code becomes harder to read, because the function which needs some speedup is no longer in the same Python source.
- If the application need to be deployed, there is usually an extra build step.

Overview of CPython

Firstly, when referring to Python, we refer to the language *not the implementation*. The most-widely used implementation of the Python programming language is CPython (Classic Python, although sometimes also referred to as C Python, since implemented in C). CPython consists of several components, most importantly a bytecode compiler and a bytecode interpreter. The bytecode compiler translates Python source code into Python bytecode. Python bytecode consists of a set of instructions for the bytecode interpreter. The bytecode interpreter (also called Python virtual machine) is executing bytecode instructions. Python bytecode is really an implementation detail of CPython, and the instruction set is not stable, i.e. the bytecode changes with every major Python version. One could perfectly write a Python interpreter which does not use bytecode at all. However, there are at least two good reasons for having bytecode as an intermediate step.

- Speed: A Python program only needs to be translated to bytecode when it is first loaded into the interpreter.
- Design: Having bytecode as an internal intermediate step simplifies the design of the (entire) interpreter, since each component (bytecode compiler and bytecode interpreter) can be individually maintained, debugged and tested.

The PyPy project

In this section, we give a brief overview of the PyPy project. The project started by writing a Python bytecode interpreter in Python itself, and grew to implement an entire Python interpreter in Python. Compared to the CPython implementation, Python takes the role of the C Code. The clear advantage of this approach is that the description of the interpreter is shorter and simpler to read, as many implementation details vanish. The obvious drawback of this approach is that this interpreter will be unbearably slow as long as it is run on top of CPython. To get to a more useful interpreter again, the PyPy project translates the high-level description of Python to a lower level one. This is done by translating the Python implementation on the Python interpreter to C source. In order to translate Python to C, the PyPy virtual machine is written in RPython. RPython is a restricted subset of Python, and the PyPy project includes a RPython translator, which can produce output in C, LLVM, and other languages.

Using the PyPy translator

The following piece of Python code shows how the translator in PyPy can be used to create a compile decorator, i.e. a decorator functions which lets the programmer easily compile a Python function:

```
from pypy.translator.interactive import Translation

class compdec:
    def __init__(self, func):
        self.func = func
        self.argtypes = None

    def __call__(self, *args):
        argtypes = tuple(type(arg) for arg in args)
        if argtypes != self.argtypes:
            self.argtypes = argtypes
            t = Translation(self.func)
            t.annotate(argtypes)
            self.cfunc = t.compile_c()

        return self.cfunc(*args)

@compdec
def is_prime(n):
    if n < 2:
        return False
    for i in xrange(2, n):
        if n%i == 0:
            return False
    return True

print sum(is_prime(n) for n in xrange(100000))
```

There are several things to note about this code:

- A decorator is only syntactic sugar.
- The decorator function is in fact a class which upon initialization receives the function to be compiled.
- When the compiled function is called, the special *__call__* method of the instance is invoked.
- A decorated function is only compiled when invoked with a new set of arguments types.
- The function which is compiled, (*is_prime* in the example) must restrict it's features to RPython, e.g. it can not contain dynamic features like Python's *eval* function.

In the above decorator example all the hard work is done by PyPy. Which includes:

- The translation of the RPython function to C.
- Invoking the C compiler to create a C extension module.
- Importing the compiled function back into the Python interpreter.

The advantage of the above approach is that the embedded function uses Python syntax and is therefore an internal part of the application. Moreover, a function can be written without even having PyPy in mind, and the compile decorator can be applied later when necessary.

A faster numpy.vectorize

Using the PyPy translator, we have implemented a function called *fast_vectorize*. It is designed to behave as NumPy's vectorize function, i.e. create a generic function object (ufunc) from a Python function. The argument types (signature) of the function need to be provided, and it is possible to provides several signatures for the same function. For each signature, the PyPy translator is invoked, to generate a C version of the function for the given signature. The UFunc object is created using the function *PyUFunc_FromFuncAndData* available through NumPy's C-API, the support code necessary to put all the pieces together is generated, and at the end *scipy.weave* is used to create the UFunc object. The figure gives a high level overview of *fast_vectorize* internals.

Here are some benchmarks in which the simple function is evaluated for a numpy array of size 10 Million using different methods:

```
def f(x):
    return 4.2 * x*x - x + 6.3
```

These benchmarks were obtained on a 2.4GHz Linux system:

	Method	Runtime (sec)	Speed vs.
1	numpy.vectorize	8.674	69.9
2	x as numpy.array	0.467	3.8
3	fast_vectorize	0.124	1.0
4	inlined	0.076	0.61

Remarks:

1. *numpy.vectorize* is slow, everything is implemented in Python, and no UFunc object is created.
2. When x is used as the array itself, the calculation is more memory-intensive, since for every step in the calculation a copy of the array is being made. For example, first 4.2 is multiplied to the array x, which results in a new array ($tmp = 4.2 * x$) which is then multiplied with the array x, which again results in a new array, and so on.
3. Here, the function f is translated to a Ufunc object which performs all steps of the calculation in compiled C code. Also worth noting is that the (inner) loop over 10 Million elements is a C loop.
4. In this example, the C function has been inlined into the inner C loop. For simplicity and clarity, this has been available in *fast_vectorize*. Also, if the function is more complicated than the one in the benchmark, the performance increase will be less significant.

When caluculation the simple quadratic function as a numpy array (2), it is also possible to rewrite the function in such a way that fewer arrays are created by doing some operations in-place, however this only created a modest speedup to 10%. A more significant speedup is achieved by rewriting the function as `f(x) = (a*x + b) * x + c`. It should be mentioned that whenever one is trying to optimize some numerical code one should *always* try to first optimize the mathematical expression or the algorithms being used *before* trying to optimize the execution of some particular piece of code.

There are many interesting applications for the PyPy translator, apart from the generation of UFunc objects. I encourage everyone interested in this subject to take a closer look at the PyPy project.

Note I am working on putting the *fast_vectorize* function in SciPy.

Acknowledgments

The author would like to thank SciPy community and Enthought for offering suggestions, assistance, and for making this work possible. In particular, I would like to thank Travis Oliphant for questioning me about how including compiled functions into Python can be done is an easy manner, and his support throughout this project. Also, I would like to thank Eric Jones for sharing his knowledge about *weave*. Thanks to Gael Varoquaux for suggesting the name *fast_vectorize*.

References

- PyPy: http://codespeak.net/pypy/dist/pypy/doc/home.html
- SWIG: http://www.swig.org/
- Pyrex: http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/
- Cython: http://cython.org/
- boost: http://www.boost.org/
- vectorize: http://scipy.org/Numpy_Example_List_With_Doc#vectorize

unPython: Converting Python Numerical Programs into C

Rahul Garg (garg1@cs.ualberta.ca) – *University of Alberta*, Canada
Jose Nelson Amaral (amaral@cs.ualberta.ca) – *University of Alberta*, Canada

unPython is a Python-to-C compiler intended for numerical Python programs. The compiler takes as input type-annotated Python source and produces C source code for an equivalent extension module. The compiler is NumPy-aware and can convert most NumPy indexing or slicing operations into C array accesses. Furthermore the compiler also allows annotating certain for-loops as parallel and can generate OpenMP code thus providing an easy way to take advantage of multicore architectures.

Introduction

Python and NumPy form an excellent environment for numerical applications. However often performance of pure Python code is not enough and the user is forced to rewrite some critical portions of the application in C. Rewriting in C requires writing glue code, manual reference count management and knowledge of Python and NumPy C APIs. This reduces the programmer productivity substantially. Moreover rewriting a module in C obscures the logic of the original Python module within a large amount of boilerplate. Thus extension modules written in C can often become very hard to maintain.

To relieve the programmer from writing C code, we present unPython. unPython is a Python to C compiler that takes as input annotated Python code and produces as output C code for an equivalent extension module. To compile a module with unPython, a programmer adds annotations, such as type declarations, to a module. The programmer then invokes unPython compiler and unPython converts the Python source into C. Annotations are added in a non-interfering way such that the annotated Python code still remains valid Python and thus can still run on CPython interpreter giving the same results as the original unannotated Python code.

The distinguishing feature of unPython is that unPython is focused on compiling numerical applications and knows about NumPy arrays. unPython therefore has knowledge of indexing and slicing operations on NumPy arrays and converts them into efficient C array accesses. The other distinguishing feature of unPython is its support for parallel loops. We have introduced a new parallel loop notation thus allowing Python programmers to take advantage of multicores and SMPs easily from within Python code. While the code runs as serial loop on the interpreter, unPython converts specially marked loops into parallel C loops. This feature is especially important since CPython has no built-in support for true concurrency and therefore all existing solutions for parallelism in Python are process based. Moreover since parallel loops are inspired from models such as OpenMP [openmp], the parallel loop will be familiar to many programmers and is easier to deal with than a general thread-and-lock-based model.

Features

1. unPython is focused on numerical applications and hence can deal with int, float, double and NumPy array datatypes. Arbitrary precision arithmetic is not supported and the basic numeric types are converted into their C counterparts. NumPy array accesses are converted into C array accesses. Currently "long" integers are not supported but will be added after transition to Python 3.

2. To compile a function, a user specifies the type signature of the function. The type signature is provided through a decorator. When running on the interpreter, the decorator simply returns the decorated function as-is. However when compiled with the unPython compiler, the decorator takes on special meaning and is seen as a type declaration. The types of all local variables are automatically inferred. To facilitate type inference, unPython requires that the type of a variable should remain constant. In Python 3, we aim to replace decorators with function annotations. An example of the current decorator-based syntax is as follows:

```
# 2 types for 2 parameters
# last type specified for return type
@unpython.type('int','int','int')
def f(x,y):
  #compiler infers type of temp to be int
  temp = x + y
  return temp
```

3. User-defined classes are supported. However multiple inheritance is not currently supported. The programmer declares the types of the member variables as well as member functions. Currently types of member variables are specified as a string just before a class declaration. Subclassing builtin types such as int, float, NumPy arrays, etc. is also not supported. Dynamic features of Python such as descriptors, properties, staticmethods, classmethods, and metaclasses are currently not supported.

4. unPython does not currently support dynamic facilities such as exceptions, iterators, generators, runtime code generation, etc.

5. Arbitrary for-loops are not supported. However simple for-loops over range or xrange are supported and are converted into efficient C counterparts.

6. Parallel loops are supported. Parallel loops are loops where each iteration of the loop can be executed independently and in any order. Thus such a loop can be speeded up if multiple cores are present. To support parallel loops, we introduce a function called prange. prange is just a normal Python function which behaves exactly like xrange on the interpreter. However, when compiling with unpython, the compiler treats it as a parallel range declaration and treats each iteration of the corresponding loop as independent. A parallel loop is converted into corresponding OpenMP declarations. OpenMP is a parallel computing industry standard supported by most modern C compilers on multicore and SMP architectures. An example of a parallel loop:

```
#assume that x is a NumPy array
#the following loop will execute in parallel
for i in unpython.prange(100):
   x[i] = x[i] + 2
```

   Under some conditions, prange loops cannot be converted to parallel C code because CPython is not thread safe. For example, if a method call is present inside a parallel loop body, then the loop is currently not parallelized and is instead compiled to a serial loop. However prange loops containing only operations on scalar numeric datatypes or NumPy arrays can usually be parallelized by the compiler.

7. Preliminary support for homogeneous lists, tuples, and dictionaries is present.

Implementation

unPython is a modular compiler implemented as multiple separate components. The compiler operates as follows:

1. A Python script uses CPython's compiler module to read a Python source file and converts the source file into an Abstract Syntax Tree (AST). AST, as the name implies, is a tree-based representation of source code. unPython uses AST throughout the compiler as the primary method of representing code.

2. The AST formed is preprocessed and dumped into a temporary file.

3. The temporary file is then read back by the core of the compiler. The core of the compiler is implemented in Java and Scala. To read the temporary file, the compiler uses a parser generated by ANTLR. The parser reads the temporary file and returns the AST read from the file.

4. Now the compiler walks over the AST to check the user-supplied type information and adds type information to each node.

5. The typed AST undergoes several transformations. The objective of each transformation is to either optimize the code represented by the AST or to convert the AST to a representation closer to C source code. Each phase is called a "lowering" of the AST because with each transformation, the AST generally becomes closer to low-level C code than high-level Python code. The term "lowering" is inspired from the Open64 [open64] compiler which also uses a tree like structure as the intermediate representation.

6. A final code generation pass takes the simplified AST and generates C code. We are looking to further split this phase so that the compiler will first generate a very low level representation before generating C code. Splitting the final code generation into two phases will allow us to easily add new backends.

Most of the compiler is implemented in Java and Scala [scala]. Scala is a statically-typed hybrid functional and object-oriented language that provides facilities such as type inference, pattern matching and higher order functions not present in Java. Scala compiles to JVM bytecodes and provides easy interoperability with Java. The choice of implementation language was affected by several factors. First, by using languages running on the JVM, we were able to utilize the standard JVM libraries like various data structures as well as third party libraries such as ANTLR and FreeMarker. Second, distribution of compiler binaries is simplified since binaries run on the JVM and are platform independent. Further, both Java and Scala usually perform much faster than Python. Finally, Scala provides language features such as pattern matching which were found to considerably simplify the code.

Experimental Results

The platform for our evaluations was AMD Phenom x4 9550 with 2 GB RAM running 32-bit Linux. GCC 4.3 was used as the backend C compiler and "-O2 -fopenmp" flags were passed to the compiler unless otherwise noted. The test codes are available at http://www.cs.ualberta.ca/~garg1/scipy08/

Recursive benchmark : Compiled vs Interpreted

The programming language shootout [shootout] is a popular benchmark suite often used to get a quick overview of speed of simple tasks in a programming language. We chose integer and floating point versions of Fibonacci and Tak functions from "recursive" benchmark as a test case. The inputs to the functions were the same as the inputs in the shootout. We chose a simple Python implementation and measured the time required by the Python interpreter to complete the benchmark. Then type annotations were

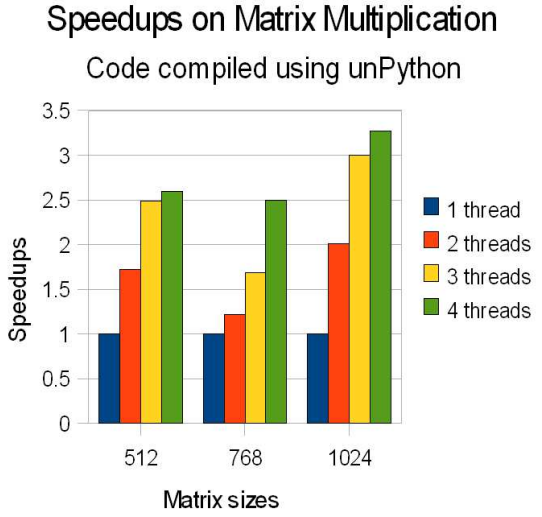
then added and the code was compiled to C using unPython.

The interpreted version finished the benchmark in 113.9 seconds while the compiled version finished in 0.77 seconds thus giving a speedup of 147x.

Matrix multiplication : Serial vs Parallel

We present experimental evaluation of the parallel loop construct "prange". We wrote a simple matrix multiplication function in Python to multiply two numpy arrays of doubles. The function was written as a 3-level loop nest with the outer loop parallelized using prange while the inner two loops were xrange.

We measured the performance of C + OpenMP code generated by unPython. For each matrix size, the number of threads was varied from 1 to 4 to obtain the execution time. The execution times for each matrix size were then divided by the execution time of 1 thread for that particular matrix size. The resulting speedups are shown in the following plot.



We also measured the execution time of a purely serial version of matrix multiplication with no parallel loops to measure the overhead of OpenMP on single thread performance. We found that the difference in execution time of the serial version and 1-thread OpenMP version was nearly zero in each case. Thus in this case we found no parallelization overhead over a serial version.

Related Work

Several other Python compilers are under development. Cython [cython] is a fork of Pyrex [pyrex] compiler. Cython takes as input a language similar to Python but with optional type declarations in a C like syntax. Pyrex/Cython produces C code for extension modules. Cython is a widely used tool and supports more Python features than unPython. Cython recently added support for efficient access to NumPy arrays using the Python buffer interface. Cython does not support parallel loops currently.

Shedskin [shedskin] is a Python to C++ compiler which aims to produce C++ code from Python code without any linking to Python interpreter. Shedskin relies on global type inference. Shedskin does not directly support numpy arrays but instead provides more efficient support for list datatype.

PyPy [pypy] is a project to implement Python in Python. PyPy project also includes a RPython to C compiler. RPython is a restricted statically typable subset of Python. PyPy has experimental support for NumPy.

Future Work

unPython is a young compiler and a work in progress. Several important changes are expected over the next year.

1. Broader support for NumPy is under development. We intend to support most methods and functions provided by the NumPy library. Support for user defined ufuncs is also planned.

2. Lack of support for exceptions is currently the weakest point of unPython. However exception support for Python is quite expensive to implement in terms of performance.

   NumPy array accesses can throw out-of-bounds exceptions. Similarly core datatypes, such as lists, can also throw many exceptions. Due to the dynamic nature of Python, even an object field access can throw an exception. Thus we are searching for a solution to deal with exceptions in a more selective manner where the user should be able to trade-off safety and performance. We are looking at prior work done in languages such as Java.

3. We intend to continue our work on parallel computing in three major directions. First we intend to investigate generation of more efficient OpenMP code. Second, we will investigate compilation to GPU architectures. Finally research is also being done on more general parallel loop support.

4. Support for the Python standard library module ctypes is also planned. ctypes allows constructing interfaces to C libraries in pure Python.

5. Research is also being conducted on more sophisticated compiler analysis such as dependence analysis.

Conclusion

The paper describes unPython, a modern compiler infrastructure for Python. unPython is a relatively young compiler infrastructure and has not yet reached its full potential. unPython has twin goals of great performance and easily accessible parallel computing. The compiler has a long way to go but we believe with community participation, the compiler will

achieve its goals over the next few years and will become a very important tool for the Python community. unPython is made available under GPLv3 at http://code.google.com/p/unpython.

References

[cython] http://cython.org
[open64] http://www.open64.net
[openmp] http://openmp.org
[pypy] http://codespeak.net/pypy
[pyrex] http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/
[scala] http://www.scala-lang.org
[shedskin] http://code.google.com/p/shedskin
[shootout] http://shootout.alioth.debian.org